AFRL-IF-RS-TR-2003-84
Final Technical Report
April 2003

AGENTS FOR PLAN MONITORING AND REPAIR

University of Southern California

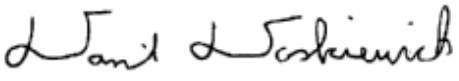
Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. G353

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

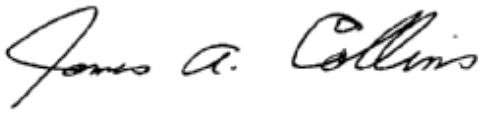
AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-84 has been reviewed and is approved for publication.

APPROVED: 

DANIEL E. DASKIEWICH
Project Engineer

FOR THE DIRECTOR: 

JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	APRIL 2003	Final Jul 98 – Jun 02

4. TITLE AND SUBTITLE
AGENTS FOR PLAN MONITORING AND REPAIR

5. FUNDING NUMBERS
C - F30602-98-2-0109
PE - 63760E
PR - AGEN
TA - T0
WU - 08

6. AUTHOR(S)
Craig A. Knoblock, Kristina Lerman, Steve Minton, and Ion Muslea

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Southern California
Information Sciences Institute
4676 Admiralty Way
Marina Del Rey California 90292-6695

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

AFRL/IFTB
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2003-84

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Daniel E. Daskiewich/IFTB/(315) 330-7731/ Daniel.Daskiewich@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

A key aspect to the successful planning and execution of tasks in an agent-enabled organization is the timely access to up-to-date information. Agents making decisions on behalf of human users need to access and verify information from multiple heterogeneous information sources, such as internal organizational databases (personal schedules, staff lists), real-time sensors (traffic and weather updates), as well as many types of public information (airline schedules, restaurant listings, etc). To address the problems of accessing and verifying information from heterogeneous sources, we developed a set of techniques for learning to recognize the content of the required information. The capability makes it possible to automatically access and maintain wrappers to extract data from online sources. It also provides semantic interoperability among software agents: to verify information from other agents, identify semantic mismatches in data, as well as automatically determine the type of information being communicated by an agent.

14. SUBJECT TERMS
Information Agent, Wrapper Learning, Wrapper Induction, Wrapper Maintenance, Ariadne, Electric Elves

15. NUMBER OF PAGES
150

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

List of Figures

1 Summary

The operation of a human organization requires dozens of everyday tasks to ensure coherence in organizational activities, to monitor the status of such activities, to gather information relevant to the organization, to keep everyone in the organization informed, etc. Teams of software agents can aid humans in accomplishing these tasks, facilitating the organization's coherent functioning and rapid response to crisis, while reducing the burden on humans. Based on this vision, we developed the Electric Elves, which has been applied to the tasks of organizing activities (such as meetings) within an organization and meeting and travel planning outside an organization.

A key aspect to the successful planning and execution of tasks in an agent-enabled organization is the timely access to up-to-date information. Agents making decisions on behalf of human users need to access and verify information from multiple heterogeneous information sources, such as internal organizational databases (personal schedules, staff lists), real-time sensors (traffic and weather updates), as well as many types of public information (airline schedules, restaurant listings, etc).

To address the problems of accessing and verifying information from heterogeneous soures, we developed a set of techniques for learning to recognize the content of the required information. The capability makes it possible to automatically access and maintain wrappers to extract data from online sources. It also provides semantic interoperability among software agents: to verify information from other agents, identify semantic mismatches in data, as well as automatically determine the type of information being communicated by an agent.

Our research addressed the shortcomings of existing systems for building wrappers for accessing online data sources. 1) Web information sources frequently change in a way that breaks the wrappers, and 2) the wrapper creation is a laborious process. Our research focused on developing machine learning techniques to build wrappers, automatically detect wrapper failure, and to recover from it.

2 Introduction

There is a tremendous amount of information available on the Web, but much of this information is not in a form that can be easily used by other applications. There are hopes that XML will solve this problem, but XML is not yet in widespread use and even in the best case it will only address the problem within application domains where the interested parties can agree on the XML schema definitions. Previous work on wrapper generation in both academic research [9, 11, 13] and commercial products (such as OnDisplay's eContent) have primarily focused on the ability to rapidly create wrappers. The previous work makes no attempt to ensure the accuracy of the wrappers over the entire set of pages of a site and provides no capability to detect failures and repair the wrappers when the underlying sources change.

We have developed the technology for rapidly building wrappers for accurately and reliably extracting data from semistructured sources. Figure 1 graphically illustrates the entire lifecycle of a wrapper. As shown in the Figure, the wrapper induction system takes a set of web pages labeled with examples of the data to be extracted. The user provides the initial set of labeled examples and the system can suggest additional pages for the user to label in order to build wrappers that are very accurate. The output of the wrapper induction system is a set of extraction rules that describe how to locate the desired information on a Web page. After the system creates a wrapper, the wrapper verification system uses the functioning wrapper to learn patterns that describe the data being extracted. If a change is detected, the system can automatically repair a wrapper by using the same patterns to locate examples on the changed pages and re-running the wrapper induction system. The details of this entire process are described in the remainder of this report.

3 Methods, Assumptions, and Procedures

3.1 Wrapper Induction

A wrapper is a piece of software that enables a semistructured Web source to be queried as if it were a database. These are sources where there is no explicit structure or schema, but there is an implicit underlying structure (for example, consider the two documents in Figure 2). Even text sources, such

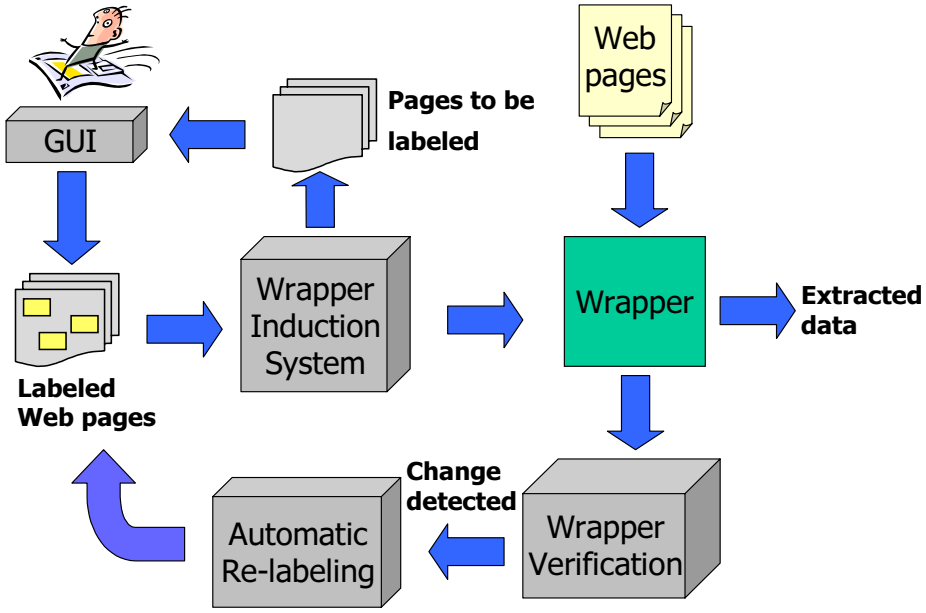


Figure 1: The Lifecycle of a Wrapper

as email messages, have some structure in the heading that can be exploited to extract the date, sender, addressee, title, and body of the messages. Other sources, such as online catalogs, have a very regular structure that can be exploited to extract the data automatically.

One of the critical problems in building a wrapper is defining a set of extraction rules that precisely define how to locate the information on the page. For any given item to be extracted from a page, one needs an extraction rule to locate both the beginning and end of that item. Since, in our framework, each document consists of a sequence of tokens (e.g., words, numbers, HTML tags, etc), this is equivalent to finding the first and last tokens of an item. The hard part of this problem is constructing a set of extraction rules that work for *all* of the pages in the source.

A key idea underlying our work is that the extraction rules are based on "landmarks" (i.e., groups of consecutive tokens) that enable a wrapper to locate the start and end of the item within the page. For example, let us consider the three restaurant descriptions `E1`, `E2`, and `E3` presented in Figure 3. In order to identify the beginning of the address, we can use the rule

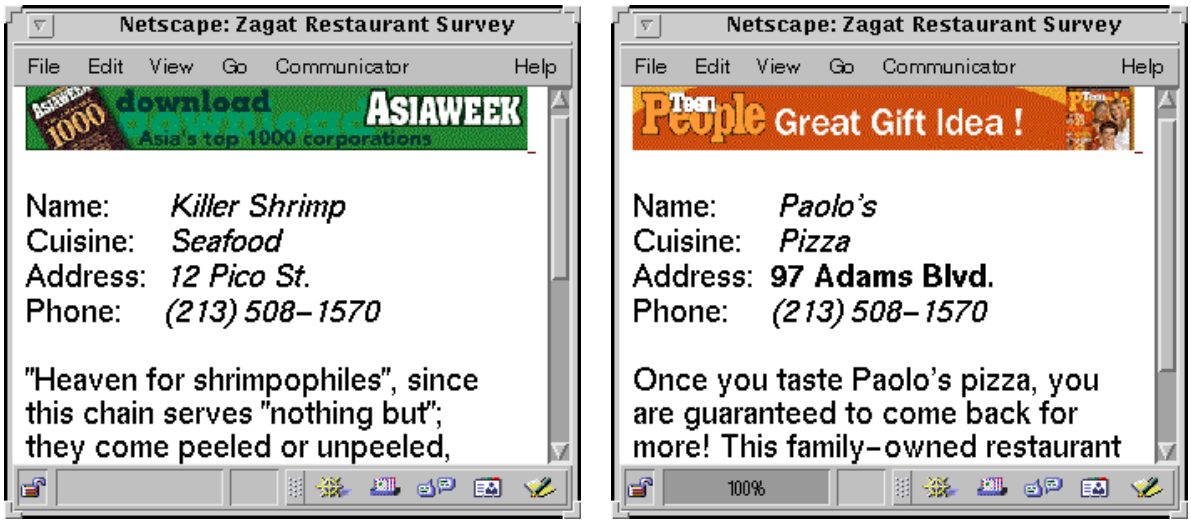


Figure 2: Two Sample Restaurant Documents From the Zagat Guide

```
E1:   ...Cuisine:<i>Seafood</i><p>Address:<i>  12 Pico St. </i><p>Phone:<i>...
E2:   ...Cuisine:<i>Thai   </i><p>Address:<i> 512 Oak Blvd.</i><p>Phone:<i>...
E3:   ...Cuisine:<i>Burgers</i><p>Address:<i> 416 Main St. </i><p>Phone:<i>...
E4:   ...Cuisine:<i>Pizza</i><p>Address:<b> 97 Adams Blvd. </b><p>Phone:<i>...
```

Figure 3: Four sample restaurant documents

R1 = $SkipTo$(`Address`) $SkipTo$(`<i>`)

which has the following meaning: start from the beginning of the document and skip every token until you find a landmark consisting of the word `Address`, and then, again, ignore everything until you find the landmark `<i>`. **R1** is called a *start rule* because it identifies the beginning of the address. One can write a similar *end rule* that finds the end of the address; for sake of simplicity, we restrict our discussion here to start rules.

Note that **R1** is by no means the only way to identfy the beginning of the address. For instance, the rules

R2 = $SkipTo$(`Address : <i>`)

R3 = $SkipTo$(`Cuisine : <i>`) $SkipTo$(`Address : <i>`)

R4 = $SkipTo$(`Cuisine : <i>` *_Capitalized_* `</i> <p> Address : <i>`)

can be also used as start rules. **R2** uses the 3-token landmark that immediately precedes the beginning of the address in examples `E1`, `E2`, and `E3`, while **R3** relies on two 3-token landmarks. Finally, **R4** is defined based on a 9-token landmark that uses the wildcard *_Capitalized_*, which is a placeholder for any capitalized alphabetic string (other examples of useful wildcards are *_Number_*, *_AllCaps_*, *_HtmlTag_*, etc).

To deal with variations in the format of the documents, our extraction rules allow the use of *disjunctions*. For example, let us assume that the addresses that are within one mile from your location appear in bold (see example `E4` in Figure 3), while the other ones are displayed as italic (e.g., `E1`, `E2`, and `E3`). We can extract all the names based on the disjunctive start rule

either $SkipTo$(`Address : <b>`)

or $SkipTo$(`Address`) $SkipTo$(`<i>`)

Disjunctive rules are *ordered lists* of individual disjuncts (i.e., decision lists). Applying a disjunctive rule is a straightforward process: the wrapper successively applies each disjunct in the list until it finds the first one that matches. Even though in our simplified examples one could have used the nondisjunctive rule

$SkipTo$(`Address :` *_HtmlTag_*),

there are many real world sources that cannot be wrapped without using disjuncts.

We have developed STALKER [19], a *hierarchical* wrapper induction algorithm that learns extraction rules based on examples labeled by the user. We have a graphical user interface that allows a user to mark up several pages on a site, and the system then generates a set of extraction rules that accurately extract the required information. Our approach uses a greedy-covering inductive learning algorithm, which incrementally builds the extraction rules from the examples.

In contrast to other approaches [9, 11, 13], a key feature of STALKER is that it is able to efficiently generate extraction rules from a small number of examples: it rarely requires more than 10 examples, and in many cases two examples are sufficient. The ability to generalize from such a small number of examples has a two-fold explanation. First, in most of the cases, the pages in a source are generated based on a fixed template that may have only a few variations. As STALKER tries to learn landmarks that are part of this template, it follows that for templates with little or no variations a handful of examples usually will be sufficient to induce reliable landmarks.

Second, STALKER exploits the *hierarchical* structure of the source to constrain the learning problem. More precisely, based on the schema of the data to be extracted, we *automatically* decompose one difficult problem (i.e., extract all items of interest) into a series of simpler ones. For instance, instead of using one complex rule that extracts all restaurant names, addresses and phone numbers from a page, we take a hierarchical approach. First we apply a rule that extracts the whole list of restaurants; then we use another rule to break the list into tuples that correspond to individual restaurants; finally, from each such tuple we extract the name, address, and phone number of the corresponding restaurant. Our hierarchical approach also has the advantage of being able to extract data from pages that contain complicated formatting layouts (e.g., lists embedded in other lists) that previous approaches could not handle (see [19] for details).

STALKER is a sequential covering algorithm that, given the training examples E, tries to learn a minimal number of *perfect disjuncts* that cover *all* examples in E. By definition, a perfect disjunct is a rule that covers at least one training example and on any example the rule matches it produces the correct result. STALKER first creates an initial set of candidate-rules C and then repeatedly applies the following three steps until it generates a perfect disjunct:

- select most promising candidate from C

- refine that candidate

- add the resulting refinements to C

Once STALKER obtains a perfect disjunct P, it removes from E all examples on which P is correct, and the whole process is repeated until there are no more training examples in E. STALKER uses two types of refinements: *landmark refinements* and *topology refinements*. The former makes the rule more specific by adding a token to one of the existing landmarks, while the latter adds a new 1-token landmark to the rule.

For instance, let us assume that based on the four examples in Figure 3, we want to learn a start rule for the address. STALKER proceeds as follows. First, it selects an example, say `E4`, to guide the search. Second, it generates a set of *initial candidates*, which are rules that consist of a single 1-token landmark; these landmarks are chosen so that they match the token that *immediately precedes* the beginning of the address in the guiding example. In our case we have two initial candidates:

R5 = $SkipTo($ `<b>` $)$

R6 = $SkipTo($ *_HtmlTag_* $)$

As the `<b>` token appears only in `E4`, **R5** does not match within the other three examples. On the other hand, **R6** matches in all four examples, even though it matches *too early* (**R6** stops as soon as it encounters an HTML tag, which happens in all four examples *before* the beginning of the address). Because **R6** has a better generalization potential, STALKER selects **R6** for further refinements.

While refining **R6**, STALKER creates, among others, the new candidates **R7**, **R8**, **R9**, and **R10** shown below. The first two are obtained via landmark refinements (i.e., a token is added to the landmark in **R6**), while the other two rules are created by topology refinements (i.e., a new landmark is added to **R6**). As **R10** works correctly on all four examples, STALKER stops the learning process and returns **R10**.

R7 = $SkipTo($: *_HtmlTag_* $)$

R8 = $SkipTo($ *_Punctuation_* *_HtmlTag_* $)$

R9 = $SkipTo($`:`$)$ $SkipTo($*_HtmlTag_*$)$

R10 = $SkipTo($`Address`$)$ $SkipTo($ *_HtmlTag_* $)$

By using STALKER, we were able to successfully wrap information sources that could not be wrapped with existing approaches (see [19] for details). In an empirical evaluation on 28 sources proposed in [13], STALKER had to learn 206 extraction rules. We learned 182 *perfect* rules (100% accurate), and another 18 rules that had an accuracy of at least 90%. In other words, only 3% of the learned rules were less that 90% accurate.

3.2 Active Learning

STALKER can do significantly better on the hard tasks (i.e., the ones for which it failed to learn perfect rules) if instead of *random* examples, the system is provided with carefully selected examples. Specifically, the most informative examples illustrate exceptional cases. However, it is unrealistic to assume that a user is willing and has the skills to browse a large number of documents in order to identify a sufficient set of examples to learn perfect extraction rules. This is a general problem that none of the existing tools address, regardless of whether they use machine learning.

To solve this problem we have developed an *active learning* approach that analyzes the set of unlabeled examples to automatically select examples for the user to label. Our approach, called *co-testing* [18], exploits the fact that there are often multiple ways of extracting the same information [4]. In the case of wrapper learning, the system can learn two different types of rules: *forward* and *backward* rules. All the rules presented above are *forward* rules: they start at the beginning of the document and go towards the end. By contrast, a *backward* rule starts at the end of the page and goes towards its beginning. For example, one can find the beginning of the addresses in Figure 3 by using one of the following backward rules:

R11 = $BackTo($`Phone`$)$ $BackTo($*_Number_*$)$

R12 = $BackTo($`Phone : <i>` $)$ $BackTo($*_Number_*$)$

The main idea behind co-testing is straightforward: after the user labels one or two examples, the system learns *both* a forward and a backward rule. Then it runs *both* rules on a given set of unlabeled pages. Whenever the rules

disagree on an example, the system considers that as an example for the user to label next. The intuition behind our approach is the following: if both rules are 100% accurate, on *every* page they must identify *the same* token as the beginning of the address. Furthermore, as the two rules are learned based on different sets of tokens (i.e., the sequences of tokens that precede and follow the beginning of the address, respectively), they are highly unlikely to make the exact same mistakes. Whenever the two rules disagree, at least one of them must be wrong, and by asking the user to label that particular example, we obtain a highly informative training example. Co-testing makes it possible to generate accurate extraction rules with a very small number of labeled examples.

To illustrate how co-testing works, consider again the examples in Figure 3. Since most of the restaurants in a city are *not* located within a 1-mile radius of one's location, it follows that most of the documents in the source will be similar to `E1`, `E2`, and `E3` (i.e., addresses shown in italic), while just a few examples will be similar to `E4` (i.e., addresses shown in bold). Consequently, it is unlikely that an address in bold will be present in a small, randomly chosen, initial training set. Let us now assume that the initial training set consists of `E1` and `E2`, while `E3` and `E4` are *not labeled*. Based on these examples, we learn the rules

Fwd-R1 = $SkipTo($`Address`$)$ $SkipTo($`<i>`$)$

Bwd-R1 = $BackTo($`Phone`$)$ $BackTo($`_Number_`$)$

Both rules correctly identify the beginning of the address for all restaurants that are more than one mile away, and, consequently, they will agree on all of them (e.g., `E3`). On the other hand, **Fwd-R1** works *incorrectly* for examples like `E4`, where it stops at the beginning of the phone number. As **Bwd-R1** is correct on `E4`, the two rules disagree on this example, and the user is asked to label it.

To our knowledge, there is no other wrapper induction algorithm that has the capability of identifying the most informative examples. In the related field of information extraction, where one wants to extract data from free text documents, researchers proposed such algorithms [21, 20], but they cannot be applied in a straightforward manner to existing wrapper induction algorithms.

We applied co-testing on the 24 tasks on which STALKER fails to learn perfect rules based on 10 random examples. To keep the comparison fair,

co-testing started with one random example and made up to 9 queries. The results were excellent: the average accuracy over all tasks improved from 85.7% to 94.2% (error rate reduced by 59.5%). Furthermore, 10 of the learned rules were 100% accurate, while another 11 rules were at least 90% accurate. In these experiments as well as in other related tests [18] applying co-testing leads to a significant improvement in accuracy without having to label more training data.

3.3 Verifying the Extracted Data

In our quest to improve our wrapper induction technology, we were faced with the problem that Web sites frequently change their format and as a result, wrappers frequently break. We realized that if a system knew enough about the type of information it was supposed to extract, then it would be possible to automatically maintain a wrapper in the face of format changes, and even automatically create wrappers for many types of websites. This idea has led to a method for automatically maintaining wrappers [17] and more recently, an initial version of an unsupervised wrapper induction system. Both of these systems use data prototypes for pattern-based bootstrapping.

Kushmerick [12] addressed the wrapper verification problem by monitoring a set of generic features, such as the density of numeric characters within a field, but this approach only detects certain types of changes. In contrast, we address this problem by applying machine learning techniques to learn a set of patterns that describe the information that is being extracted from each of the relevant fields. Since the information for even a single field can vary considerably, the system learns the statistical distribution of the patterns for each field. Wrappers can be verified by comparing the patterns of data returned to the learned statistical distribution. When a significant difference is found, an operator can be notified or we can automatically launch the wrapper repair process, which is described in the next section.

Data prototypes are patterns of specific tokens and syntactic token types that describe the common beginnings and endings of a set of examples of data [17]. For example, a set of street addresses - "4676 Admiralty Way", "220 Lincoln Boulevard" and "998 South Robertson Boulevard" - start with a pattern "Number Caps" and end with "Boulevard" or "Caps." The starting and ending patterns together define a data prototype. We have developed DataPro - an algorithm that learns data prototypes from positive examples of the data field. DataPro finds all statistically significant sequences of tokens

- a sequence of tokens is significant if it occurs more frequently than would be expected if the tokens were generated randomly and independently of one another. We begin by estimating the baseline probability of a token's occurrence from the proportion of all tokens in the examples for the data field that are of that type. Suppose we have already found a significant token sequence - e.g., the pattern consisting of the single token "Number"- in a set of street addresses such as the ones above, and want to determine whether the more specific pattern, "Number Caps", is also a significant pattern. Knowing the probability of occurrence of type Caps, we can compute how many times Caps can be expected to follow Number completely by chance. If we observe a considerably greater number of these sequences, we conclude that the longer pattern is significant.

The learned patterns represent the structure, or format, of the field as a sequence of words and wildcards [14]. Wildcards represent syntactic categories to which words belong — alphabetic, numeric, capitalized, etc. — and allow for multi-level generalization. For complex fields, and for purposes of information extraction, it is sufficient to use only the starting and ending patterns as the description of the data field. For example, a set of street addresses — **12 Pico St.**, **512 Oak Blvd.**, **416 Main St.** and **97 Adams Blvd.** — all start with a pattern **(_*Number*_ _*Capitalized*_)** and end with **(Blvd.)** or **(St.)**. We refer to the starting and ending patterns together as the *data prototype* of the field.

The problem of learning a description of a field (class) from a set of labeled examples can be posed in two related ways: as a classification or a conservation task. If negative examples are present, the classification algorithm learns a *discriminating* description. When only positive examples are available, the conservation task learns a *characteristic* description, *i.e.* one that describes many of the positive examples but is highly unlikely to describe a randomly generated example. Because an appropriate set of negative examples not available in our case, we chose to frame the learning problem as a conservation task. The algorithm we developed, called DataPro [14], finds all statistically significant starting and ending patterns in a set of positive examples of the field. A pattern is said to be significant if it occurs more frequently than would be expected by chance if the tokens were generated randomly and independently of one another. Our approach is similar to work on grammar induction [5, 10], but our pattern language is better suited to capturing the regularities in small data fields (as opposed to languages).

The algorithm operates by building a prefix tree, where each node cor-

responds to a token whose position in the sequence is given by the node's depth in the tree. Every path through the tree starting at the root node is a significant pattern found by the algorithm.

The algorithm grows the tree incrementally. Adding a child to a node corresponds to extending the node's pattern by a single token. Thus, each child represents a different way to specialize the pattern. For example, when learning city names, the node corresponding to "New" might have three children, corresponding to the patterns "New Haven", "New York" and "New _*Capitalized*_". A child node is judged to be significant with respect to its parent node if the number of occurrences of the child pattern is sufficiently large, given the number of occurrences of the parent pattern and the baseline probability of the token used to extend the parent pattern. A pruning step insures that each child node is also significant given its more specific siblings. For example, if there are 10 occurrences of "New Haven", and 12 occurrences of "New York", and both of these are judged to be significant, then "New _*Capitalized*_" will be retained only if there are significantly more than 22 examples that match "New _*Capitalized*_". Similarly, once the entire tree has been expanded, the algorithm includes a pattern extraction step that traverses the tree, checking whether the pattern "New" is still significant given the more specific patterns "New York", "New Haven" and "New _*Capitalized*_". In other words, DataPro decides whether the examples described by "New" but not by any of the longer sequences can be explained by the null hypothesis.

The patterns learned by DataPro lend themselves to the data validation task and, specifically, to wrapper verification. A set of queries is used to retrieve HTML pages from which the wrapper extracts some training examples. The learning algorithm finds the patterns that describe the common beginnings and endings of each field of the training examples. In the verification phase, the wrapper generates a test set of examples from pages retrieved using the same or similar set of queries. If the patterns describe statistically the same (at a given significance level) proportion of the test examples as the training examples, the wrapper is judged to be extracting correctly; otherwise, it is judged to have failed.

Once we have learned the patterns, which represent our expectations about the format of data, we can then configure the wrappers to verify the extracted data before the data is sent, immediately after the results are sent, or on some regular frequency. The most appropriate verification method depends on the particular application and the tradeoff between response time

and data accuracy.

We monitored 27 wrappers (representing 23 distinct Web sources) over a period of several months. For each wrapper, the results of 15-30 queries were stored periodically. All new results were compared with the last correct wrapper output (training examples). A manual check of the results revealed 37 wrapper changes out of the total 443 comparisons. The verification algorithm correctly discovered 35 of these changes. The algorithm incorrectly decided that the wrapper has changed in 40 cases.

Figure 4: An Example of the Reinduction Process

3.4 Automatically Repairing Wrappers

We have built a system that uses data prototypes to re-induce a wrapper when a site's format changes [16]. The system takes a working wrapper, and learns data prototypes for each field extracted. If the wrapper later breaks, the system can frequently find enough examples of each field on new sample pages to automatically seed the STALKER induction algorithm.

Most changes to Web sites are largely syntactic and are often minor formatting changes or slight reorganizations of a page. Since the content of the fields tend to remain the same, we exploit the patterns learned for verifying the extracted results to locate correct examples of the data field on new pages. Once the required information has been located, the pages are automatically re-labeled and the labeled examples are re-run through the inductive learning process to produce the correct rules for this site.

Specifically, the wrapper reinduction algorithm takes a set of training examples and a set of pages from the same source and uses machine learning techniques to identify examples of the data field on new pages. First, it learns the starting and ending patterns that describe each field of the training examples. Next, each new page is scanned to identify all text segments that begin with one of the starting patterns and end with one of the ending patterns. Those segments, which we call candidates, that are significantly longer or shorter than the training examples are eliminated from the set, often still leaving hundreds of candidates per page. The candidates are then clustered to identify subgroups that share common features. The features used in clustering are the candidate's relative position on the page, adjacent landmarks, and whether it is visible to the user. Each group is then given a score based on how similar it is to the training examples. We expect the highest ranked group to contain the correct examples of the data field.

Figure 4 shows a actual example of a change to Amazon's site and how our system automatically adapts to the change. The top frame shows an example of the original site, the extraction rule for a book title, and the extracted results from the example page. The middle frame shows the source and the incorrectly extracted result after the title's font and color were changed. And the bottom frame shows the result of the automatic reinduction process with the corrected rule for extracting the title.

We evaluated the algorithm by using it to extract data from Web pages for which correct output is known. The output of the extraction algorithm is a ranked list of clusters for every data field being extracted. Each cluster

is checked manually, and it is judged to be correct if it contains only the complete instances of the field, which appear in the correct context on the page. For example, if extracting a city of an address, we only want to extract those city names that are part of an address.

We ran the extraction algorithm for 21 distinct Web sources, attempting to extract 77 data fields from all the sources. In 62 cases the top ranked cluster contained correct complete instances of the data field. In eight cases the correct cluster was ranked lower, while in six cases no candidates were identified on the pages. The most closely related work is that of Cohen [8], which uses an information retrieval approach to relocating the information on a page. The approach was not evaluated on actual changes to Web pages, so it is difficult to assess whether this approach would work in practice.

3.5 Automatically Building Wrappers

More recently, we used data prototypes to automatically create wrappers that extract data from pages with arbitrarily structured lists and group the data into rows and columns [15] (assuming the list is not already formatted using explicit HTML lists or tables, in which case the task is trivial). This is useful when we want to rapidly create a wrapper without asking the user to label data on sample pages.

Our algorithm uses only some very general assumptions about the structure of a page. Data in a list is not always laid out in a grid-like pattern; however, it is almost always arranged with a certain amount of regularity we can exploit. For example, consider a white pages listing of names and addresses. The elements are arranged in the same order for each listing, so if the address follows the name in one row, it usually follows the name in all other rows. While such principles may not apply universally, we have found them to be valid for many online sources we studied, including airport listings, online catalogs, hotel directories, etc.

The algorithm operates by first extracting all visible (nonHTML) text. A "chunk" of text is defined to be a string of text with HTML tokens to the left and right. The task is then to produce a table from these chunks. As we mentioned above, we expect all data in the same column to be of the same type and therefore have the same or similar format. The algorithm takes all the chunks and learns data prototypes to capture these regularities. It then clusters the chunks according to the separators on each side and the data prototypes that it matches. Each resulting cluster usually consists of

the data from a single column. From this point, it is straightforward to heuristically construct the columns and rows in a table.

3.6 Electric Elves

As part of this project, we participated in the unified Electric Elves project [6, 7, 3] which was a joint effort of several groups working at ISI. This project included our group, and those headed by Milind Tambe, Yolanda Gil and Hans Chalupsky. The purpose of the joint effort was to integrate the various agent technologies at ISI into a framework that will enable the rapid deployment of novel agent applications.

Many activities of a human organization are well-suited for software agents, which can devote significant resources to perform these tasks, thus reducing the burden on humans. Indeed, teams of such software agents could assist all organizations, including disaster response organizations, corporations, the military, universities and research institutions.

The operation of a human organization involves dozens of critical everyday tasks to ensure coherence in organizational activities, to monitor the status of such activities, to obtain information relevant to the organization, to keep everyone in the organization informed, etc. These activities are often well-suited for software agents, which can devote significant resources to perform these tasks, thus reducing the burden on humans. Indeed, teams of such software agents, including proxy agents that act on behalf of humans, would enable organizations to act coherently, to attain their mission goals robustly, to react to crises swiftly, and to adapt to events dynamically. Such agent teams could assist all organizations, including the military, civilian disaster response organizations, corporations, and universities and research institutions.

Within an organization, we envision agents assisting in *all* of its day-to-day functioning. For a research institution, agents may facilitate activities such as meeting (re)scheduling, selecting presenters for research meetings, composing papers, developing software and deploying people and equipment for out-of-town demonstrations. For a disaster response organization, agents may facilitate the teaming of people and equipment to rapidly respond to crises (e.g., earthquakes), to monitor the progress of any such for rapid response, etc. To accomplish such goals, each person in an organization will have an agent proxy. For instance, if an organizational crisis requires an urgent deployment of a team of people and equipment, then agent proxies could

dynamically volunteer for team membership on behalf of the people or resources they represent, while also ensuring that the selected team collectively possesses sufficient resources and capabilities. The proxies must also manage efficient transportation of such resources, the monitoring of the progress of individual participants and of the mission as a whole, and the execution of corrective actions when goals appear to be endangered.

Based on the above vision, we have developed a system called *Electric Elves* that applies agent technology in service of the day-to-day activities of the Intelligent Systems Division of USC/ISI. Electric Elves is a system of about 15 agents, including nine proxies for nine people, plus two different matchmakers, one flight tracker and one scheduler running continuously for past several months.

The primary research issues that we focused on within the Elves project was access to online sources, detection of changes to those sources, automatic adaptation to those changes, and, ultimately, automatic access to previously unseen sources. These problems were central to the Electric Elves effort since reliable access to data from online sources and other agents is a critical problem in building a reliable network of personal agents.

The Electric Elves project uses Ariadne wrappers to extract information from Web sources, such as online bibliographic databases, airline schedules, and restaurant reviews. Currently, wrapper creation is a semi-automated process: the user marks up several examples of the data to be extracted from the Web page, and a learning system learns the appropriate extraction rules. Our research addresses two of the shortcomings of the present wrapper creation system. Web-based information sources frequently change in a way that breaks the wrapper's extraction rules (e.g., Web site changes its layout). We developed automatic methods to detect wrapper failures and to recover gracefully from them, i.e., to learn new extraction rules with minimal user intervention.

4 Results and Discussion

Our work addresses the complete wrapper creation problem, which includes:

- building wrappers by example,
- ensuring that the wrappers accurately extract data across an entire collection of pages,

- verifying a wrapper to avoid failures when a site changes,
- and automatically repair wrappers in response to changes in layout or format.

Our main technical contribution is in the use of machine learning to accomplish all of these tasks. Essentially, our approach takes advantage of the fact that web pages have a high degree of regular structure. By analyzing the regular structure of example pages, our wrapper induction process can detect landmarks that enable us to extract desired fields. After we developed an initial wrapper induction process we realized that the accuracy of the induction method can be improved by simultaneously learning "forward" and "backward" extraction rules to identify exception cases. Again, what makes this possible is the regularities on a page that enable us to identify landmarks both before a field and after a field.

Our approach to automatically detecting wrapper breakages and repairing them capitalizes on the regular structure of the extracted fields themselves. Once a wrapper has been initially created, we can use it to obtain numerous examples of the fields. This enables us to profile the information in the fields and obtain structural descriptions that we can use during monitoring and repair. Essentially this is a bootstrapping approach. Given the initial examples provided by the user, we first learn a wrapper. Then we use this wrapper to obtain many more examples which we then analyze in much greater depth. Thus by leveraging a few human-provided examples, we end up with a highly scalable system for wrapper creation and maintenance.

The following publications were funded completely or in part by this project:

1. David N. Allsopp, Patrick Beautement, Michael Kirton, Jeffrey M. Bradshaw, Niranjan Suri, Edmund H. Durfee, Craig A. Knoblock, Austin Tate, and Craig W. Thompson. Coalition agents experiment: Multiagent cooperation in international coalitions. *IEEE Intelligent Systems*, 17(3):26–35, May/June 2002.

2. Hans Chalupsky, Yolanda Gil, Craig A. Knoblock, Kristina Lerman, Jean Oh, David V. Pynadath, Thomas A. Russ, and Milind Tambe. Electric elves: Agent technology for supporting human organizations. *AI Magazine*, 23(2):11–24, Summer 2002.

3. Kristina Lerman, Steven Minton, and Craig A. Knoblock. Wrapper maintenance: A machine learning approach. *To appear in the Journal of Artificial Intelligence Research*, 2002.

4. Sheila Tejada, Craig A. Knoblock, and Steven Minton. Learning domain-independent string transformation weights for high accuracy object identification. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2002)*, Edmonton, Alberta, Canada, 2002.

5. Sheila Tejada. *Learning Object Identification Rules for Information Integration*. PhD thesis, Department of Computer Science, University of Southern California, 2002.

6. Jose Luis Ambite, Greg Barish, Craig A. Knoblock, Maria Muslea, Jean Oh, and Steven Minton. Getting from here to there: Interactive planning and agent execution for optimizing travel. In *Proceedings of the Fourteenth Conference on Innovative Applications of Artificial Intelligence (IAAI-2002)*, pages 862–869, Edmonton, Alberta, Canada, 2002.

7. Ion Muslea, Steven Minton, and Craig A. Knoblock. Active + semi-supervised learning = robust multi-view learning. In *Proceedings of the 19th International Conference on Machine Learning (ICML-2002)*, pages 435–442, Sydney, Australia, 2002.

8. Ion Muslea, Steven Minton, and Craig A. Knoblock. Adaptive view validation: A first step towards automatic view detection. In *Proceedings of the 19th International Conference on Machine Learning (ICML-2002)*, pages 443–450, Sydney, Australia, 2002.

9. Greg Barish and Craig A. Knoblock. An efficient and expressive language for information gathering on the web. In *Proceedings of the AIPS-2002 Workshop on Is there life after operator sequencing? – Exploring real world planning*, pages 5–12, Tolouse, France, 2002.

10. D.N. Allsopp, P. Beautement, J.M. Bradshaw, E.H. Durfee, M. Kirton, C.A. Knoblock, N. Suri, A. Tate, and C.W. Thompson. Coalition agents experiment: Multi-agent co-operation in an international coalition setting. In *Proceedings of the Second International Conference on*

Knowledge Systems for Coalition Operations (KSCO-2002), Toulouse, France, April 2002.

11. Craig A. Knoblock, Kristina Lerman, Steven Minton, and Ion Muslea. Accurately and reliably extracting data from the web: A machine learning approach. In P.S. Szczepaniak, J. Segovia, J. Kacprzyk, and L.A. Zadeh, editors, *Intelligent Exploration of the Web*. Springer-Verlag, Berkeley, CA, forthcoming.

12. Naveen Ashish, Craig A. Knoblock, and Cyrus Shahabi. Selectively materializing data in mediators by analyzing user queries. *International Journal of Cooperative Information Systems*, 11(1), March 2002.

13. Jose Luis Ambite and Craig A. Knoblock. Planning by rewriting. *Journal of Artificial Intelligence Research*, 15:207–261, 2001.

14. Sheila Tejada, Craig A. Knoblock, and Steven Minton. Learning object identification rules for information integration. *Information Systems*, 26(8), 2001.

15. Hans Chalupsky, Yolanda Gil, Craig A. Knoblock, Kristina Lerman, Jean Oh, David V. Pynadath, Thomas A. Russ, and Milind Tambe. Electric elves: Applying agent technology to support human organizations. In *Proceedings of the Conference on Innovative Applications of Artificial Intelligence*, 2001.

16. Kristina Lerman, Craig A. Knoblock, and Steven Minton. Automatic data extraction from lists and tables in web sources. In *Proceedings of the IJCAI 2001 Workshop on Adaptive Text Extraction and Mining*, Seattle, WA, 2001.

17. Craig A. Knoblock, Jose Luis Ambite, Steven Minton, Cyrus Shahabi, Mohammad Kolahdouzan, Maria Muslea, Jean Oh, and Snehal Thakkar. Integrating the world: The worldinfo assistant. In *Proceedings of the 2001 International Conference on Artificial Intelligence (IC-AI 2001)*, Las Vegas, NV, 2001.

18. Craig A. Knoblock, Steven Minton, Jose Luis Ambite, Maria Muslea, Jean Oh, and Martin Frank. Mixed-initiative, multi-source information assistants. In *Proceedings of the World Wide Web Conference*, Hong Kong, May 2001.

19. Craig A. Knoblock, Steven Minton, Jose Luis Ambite, Naveen Ashish, Ion Muslea, Andrew G. Philpot, and Sheila Tejada. The ariadne approach to web-based information integration. *International Journal of Cooperative Information Systems (IJCIS), Special Issue on Intelligent Information Agents: Theory and Applications*, 10(1/2):145–169, 2001.

20. Jose Luis Ambite, Craig A. Knoblock, Ion Muslea, and Andrew Philpot. Compiling source descriptions for efficient and flexible information integration. *Journal of Intelligent Information Systems*, 16(2):149–187, March 2001.

21. Martin Frank, Maria Muslea, Jean Oh, Steve Minton, and Craig Knoblock. An intelligent user interface for mixed-initiative multi-source travel. In *Proceedings of the ACM International Conference on Intelligent User Interfaces*, Santa Fe, New Mexico, January 2001.

22. Craig A. Knoblock, Kristina Lerman, Steven Minton, and Ion Muslea. Accurately and reliably extracting data from the web: A machine learning approach. *IEEE Data Engineering Bulletin*, 23(4), December 2000.

23. Greg Barish, Daniel DiPasquo, Craig A. Knoblock, and Steven Minton. Dataflow plan execution for software agents. In *Proceedings of the Fourth International Conference on Autonomous Agents (Agents-2000)*, Barcelona, Spain, 2000.

24. Greg Barish, Daniel DiPasquo, Craig A. Knoblock, and Steven Minton. A dataflow approach to agent-based information management. In *Proceedings of the 2000 International Conference on Artificial Intelligence (IC-AI 2000)*, Las Vegas, NV, 2000.

25. Greg Barish, Craig A. Knoblock, Yi-Shin Chen, Steven Minton, Andrew Philpot, and Cyrus Shahabi. The theaterloc virtual application. In *Proceedings of Twelfth Annual Conference on Innovative Applications of Artificial Intelligence (IAAI-2000)*, Austin, Texas, 2000.

26. Jose Luis Ambite, Craig A. Knoblock, and Steven Minton. Learning plan rewriting rules. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, Breckenridge, CO, 2000.

27. Jose Luis Ambite and Craig A. Knoblock. Flexible and scalable cost-based query planning in mediators: A transformational approach. *Artificial Intelligence Journal*, 118(1-2):115–161, April 2000.

28. Greg Barish, Craig A. Knoblock, Yi-Shin Chen, Steven Minton, Andrew Philpot, and Cyrus Shahabi. Theaterloc: A case study in building an information integration application. In *Proceedings of the IJCAI-99 Workshop on Intelligent Information Integration*, Stockholm, Sweden, 1999.

29. Greg Barish, Craig A. Knoblock, Daniel DiPasquo, and Steven Minton. An efficient plan execution system for information management agents. In *Proceedings of the ACM CIKM'99 Workshop on Web Information and Data Management (WIDM)*, Kansas City, MO, 1999.

30. Jose Luis Ambite and Craig A. Knoblock. Flexible and scalable query planning in distributed and heterogeneous environments. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, Pittsburgh, PA, 1998.

5 Conclusions

Our work on wrapper creation and maintenance and semantic interoperability plays a central role in providing access to real-world data. This work provided a key capability for the Electric Elves project. The results of this research were also applied in several different applications. First, we have applied our wrapper creation and maintenance in the context of the CoAX TIE [1, 2] in order to provide access to coalition data. In a coalition environment, the capability to build and maintain wrappers is important for rapidly accessing data from both coalition partners and open sources.

Second, we have applied our work on wrapper maintenance to a project for Special Operations for automatically building Target Intelligence Packages from open sources. The ability to create, verify, and maintain access to online data sources provides a critical capability to automatically gather data for military planning and intelligence. This application has received enthusiastic reviews from various intelligence analysts and military planners. Today Target Intelligence Packages are laboriously constructed by hand. Using the

wrappers, we can automatically populate a large portion of a Target Intelligence Package in a matter of minutes. The system uses a variety of open sources and integrates this information with imagery, maps, and vector data that are stored on local servers.

Finally, much of the technology developed under this contract has been licensed to Fetch Technologies (www.fetch.com), which has turned it into commercial products for creating and building agents for extracting online data. An important customer for Fetch is the US intelligence community, which is currently using these products for their own intelligence gathering efforts.

References

[1] David N. Allsopp, Patrick Beautement, Michael Kirton, Jeffrey M. Bradshaw, Niranjan Suri, Edmund H. Durfee, Craig A. Knoblock, Austin Tate, and Craig W. Thompson. Coalition agents experiment: Multiagent cooperation in international coalitions. *IEEE Intelligent Systems*, 17(3):26–35, May/June 2002.

[2] D.N. Allsopp, P. Beautement, J.M. Bradshaw, E.H. Durfee, M. Kirton, C.A. Knoblock, N. Suri, A. Tate, and C.W. Thompson. Coalition agents experiment: Multi-agent co-operation in an international coalition setting. In *Proceedings of the Second International Conference on Knowledge Systems for Coalition Operations (KSCO-2002)*, Toulouse, France, April 2002.

[3] Jose Luis Ambite, Greg Barish, Craig A. Knoblock, Maria Muslea, Jean Oh, and Steven Minton. Getting from here to there: Interactive planning and agent execution for optimizing travel. In *Proceedings of the Fourteenth Conference on Innovative Applications of Artificial Intelligence (IAAI-2002)*, pages 862–869, AAAI Press, Menlo Park, CA, 2002.

[4] A. Blum and T. Mitchell. Combining labeled and unlabeled data with co-training. In *Proc. of the 1988 Conference on Computational Learning Theory*, pages 92–100, 1998.

[5] R. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In *Lecture Notes In Computer Science*, page 862, 1994.

[6] Hans Chalupsky, Yolanda Gil, Craig A. Knoblock, Kristina Lerman, Jean Oh, David V. Pynadath, Thomas A. Russ, and Milind Tambe. Electric elves: Agent technology for supporting human organizations. *AI Magazine*, 23(2):11–24, Summer 2002.

[7] Hans Chalupsky, Yolanda Gil, Craig A. Knoblock, Kristina Lerman, Jean Oh, David V. Pynadath, Thomas A. Russ, and Milind Tambe. Electric elves: Applying agent technology to support human organizations. In *Proceedings of the Conference on Innovative Applications of Artificial Intelligence*, 2001.

[8] W. Cohen. Recognizing structure in web pages using similarity queries. In *Proc. of the 16th National Conference on Artificial Intelligence AAAI-1999*, pages 59–66, 1999.

[9] D. Freitag and N. Kushmerick. Boosted wrapper induction. In *Proc. of the 17th National Conference on Artificial Intelligence AAAI-2000*, pages 577–583, 2000.

[10] T. Goan, N. Benson, and O. Etzioni. A grammar inference algorithm for the world wide web. In *Proc. of the AAAI Spring Symposium on Machine Learning in Information Access*, 1996.

[11] C. Hsu and M. Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Journal of Information Systems*, 23(8):521–538, 1998.

[12] N. Kushmerick. Regression testing for wrapper maintenance. In *Proc. of the 16th National Conference on Artificial Intelligence AAAI-1999*, pages 74–79, 1999.

[13] N. Kushmerick. Wrapper induction: efficiency and expressiveness. *Artificial Intelligence Journal*, 118(1-2):15–68, 2000.

[14] K. Lerman and S. Minton. Learning the common structure of data. In *Proc. of the 17th National Conference on Artificial Intelligence AAAI-2000*, pages 609–614, 2000.

[15] Kristina Lerman, Craig A. Knoblock, and Steven Minton. Automatic data extraction from lists and tables in web sources. In *Proceedings of the IJCAI 2001 Workshop on Adaptive Text Extraction and Mining*, Seattle, WA, 2001.

[16] Kristina Lerman and Steven Minton. Learning the common structure of data. In *Proceedings of the 17th National Conference on Artificial Intelligence*, pages 609–614, 2000.

[17] Kristina Lerman, Steven Minton, and Craig A. Knoblock. Wrapper maintenance: A machine learning approach. *To appear in the Journal of Artificial Intelligence Research*, 2002.

[18] I. Muslea, S. Minton, and C. Knoblock. Co-testing: Selective sampling with redundant views. In *Proc. of the 17th National Conference on Artificial Intelligence AAAI-2000*, pages 621–626, 2000.

[19] I. Muslea, S. Minton, and C. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Journal of Autonomous Agents and Multi-Agent Systems*, 2001. (to appear).

[20] S. Soderland. Learning extraction rules for semi-structured and free text. *Machine Learning*, 34:233–272, 1999.

[21] C. Thompson, M. Califf, and R. Mooney. Active learning for natural language parsing and information extraction. In *Proc. of the 16th International Conference on Machine Learning ICML-99*, pages 406–414, 1999.

Appendix A:

Electric Elves: Applying Agent Technology to Support Human Organizations *†

Hans Chalupsky, Yolanda Gil, Craig A. Knoblock, Kristina Lerman,
Jean Oh, David V. Pynadath, Thomas A. Russ, Milind Tambe
Information Sciences Institute and Computer Science Department
University of Southern California
4676 Admiralty Way, Marina del Rey, CA 90292

Abstract

The operation of a human organization requires dozens of everyday tasks to ensure coherence in organizational activities, to monitor the status of such activities, to gather information relevant to the organization, to keep everyone in the organization informed, etc. Teams of software agents can aid humans in accomplishing these tasks, facilitating the organization's coherent functioning and rapid response to crises, while reducing the burden on humans. Based on this vision, this paper reports on *Electric Elves*, a system that has been operational, 24/7, at our research institute since June 1, 2000.

Tied to individual user workstations, fax machines, voice, mobile devices such as cell phones and palm pilots, Electric Elves has assisted us in routine tasks, such as rescheduling meetings, selecting presenters for research meetings, tracking people's locations, organizing lunch meetings, etc. We discuss the underlying AI technologies that led to the success of Electric Elves, including technologies devoted to agent-human interactions, agent coordination, accessing multiple heterogeneous information sources, dynamic assignment of organizational tasks, and deriving information about organization members. We also report the results of deploying Electric Elves in our own research organization.

*Appeared in the Proceedings of the Thirteenth Innovative Applications of Arti£cial Intelligence Conference (IAAI-2001), Seattle, WA, 2001.

1 Introduction

Many activities of a human organization are well-suited for software agents, which can devote significant resources to perform these tasks, thus reducing the burden on humans. Indeed, teams of such software agents could assist all organizations, including disaster response organizations, corporations, the military, universities and research institutions.

Based on the above vision, we have developed a system called *Electric Elves* that applies agent technology in service of the day-to-day activities of the Intelligent Systems Division of USC/ISI. Electric Elves is a system of about 15 agents, including nine proxies for nine people, plus two different matchmakers, one flight tracker and one scheduler running continuously for past several months. This paper discusses the tasks performed by the system, the research challenges it faced and its use of AI technology in overcoming those challenges.

One key contribution of this paper is understanding the challenges faced in deploying agents to support organizations. In particular, the complexity inherent in human organizations complicates all of the tasks agents must perform. First, since agents must interact with humans, issues of *adjustable autonomy* become critical. In particular, agents acting as proxies for people must automatically adjust their own autonomy, e.g., avoiding critical errors, possibly by letting people make important decisions while autonomously making the more routine decisions. Second, to accomplish their goals, agents must be provided reliable access to information. Third, people have a wide variety of capabilities, interests, preferences and engage in many different tasks. To enable teaming among such people for crisis response or other organizational tasks, agents acting as their proxies must represent and reason with such capabilities and interests. We thus require powerful matchmaking capabilities to match both interests and capabilities. Fourth, coordination of all of these different agents, including proxies, is itself a significant research challenge. Finally, the entire agent system must scale-up: (i) it must scale-up in the sense of running continually 24 hours a day 7 days a week (24/7) for months at a time; (ii) it must scale-up in the number of agents to support large-scale human organizations.

2 The Electric Elves

In the Electric Elves project we have developed technology and tools for deploying agents into human organizations to help with organizational tasks. We de-

scribe the application of the Electric Elves to two classes of tasks. First, we describe the problem of coordinating activities within an individual research project. These tasks must be tightly coordinated and a significant amount of information is known in advance about the participants and their goals and capabilities. Second, in order to demonstrate the capabilities of the system in a more open environment, we applied the system to the problem of meeting planning with participants outside the organization where some of the necessary information about participants is not known in advance.

2.1 Coordinating Project Activities

Our agents help coordinate the everyday activities of a research project: they keep the project running smoothly, rescheduling meetings when someone is delayed, ordering food for meetings or if someone has to work late, and identifying speakers for research meetings. Each person in the project is assigned their own personal proxy agent, which represents that person to the agent system.

A proxy agent keeps track of a project member's current location using several different information sources, including their calendar, Global Position System (GPS) device when outside of the building (Fig. 1), infrared communications within the building, and computer activity. When a proxy agent notices that someone is not attending a scheduled meeting or that they are too far away to make it to a scheduled meeting in time, then their agent sends them a request using a wireless device (i.e., a cell phone or Palm Pilot) asking if they want to cancel the meeting, delay the meeting, or have the meeting proceed without them. If a user responds, their decision is communicated to the other participants of the scheduled meeting. If they are unable to respond, the agent must make a decision autonomously.

Figure 1: A Palm VII PDA with GPS receiver

For weekly project meetings, the agents coordinate the selection of the presenter and arrange food for the meetings. Once a week an auction is held where all of the meeting participants are asked about their capability and willingness to present at the next meeting. Then the system compiles the bids, selects a presenter, and notifies all of the attendees who will be presenting at the next project meeting. The agents also arrange food for lunch meetings. They order from a set of nearby

restaurants, select meals that were highly rated by others, and fax the orders directly to the restaurant with instructions for delivery. We have begun relying on our agents so heavily to order lunch that one local "Subway" restaurant owner even remarked: *"...more and more computers are getting to order food...so we might have to think about marketing [to them]."*

Some of the technical challenges in building this application are in determining how much autonomy the agents should assume on behalf of the user, dynamically building agent teams, determining how to assign the organizational tasks (e.g., presentations), and providing access to online data such as calendars and restaurants.

2.2 Organizing External Meetings

To demonstrate how the technology supports less structured environments, we also applied the Electric Elves to the task of planning and coordinating ad hoc meetings at conferences and workshops involving individuals across different organizations. The system identifies people that have similar research interests, coordinates scheduling a meeting with those people, locates a suitable restaurant for a meeting that takes into account dietary constraints, and makes a reservation using an online reservation service.

To identify individuals with related interests, the agents use an online bibliography service that provides a list of the papers written by an individual. When a person is going to a meeting, their agent can check an online source to locate individuals going to the same meeting and then build a model of the research interests of the different participants based on their publications. Using this information, the user selects the participants for the meeting and the agent sends out an invitation to each of the potential attendees.

Once the agent has finalized the set of participants for a meeting, it selects an appropriate place to have the meeting. It does this by checking for any known dietary restrictions and uses that information to identify suitable cuisine types. Next, the agent goes out to an online restaurant reservation site to find the set of restaurants closest to the given location and matches up these restaurants with a restaurant review site to select the high-quality restaurants. The user selects from a small set of close, highly-recommended restaurants and the agent then makes a reservation for the meeting using the online reservation system.

This application highlights two additional technical challenges: gathering information about people from other organizations and ensuring the robustness of the interaction with online sources that change frequently.

3 Underlying Technologies

In this section we describe how we addressed some of the technical challenges, namely the issues of interacting with human users within an organization, providing reliable access to organization-related data, dynamic assignment of organizational tasks, deriving knowledge about the participants in an organization, and coordination of agent teams.

3.1 Agent Interactions with Human Users

Electric Elves agents must often take actions on behalf of the human users. Specifically, a user's agent proxy (named "Friday" after Robinson Crusoe's servant and companion) can take autonomous actions to coordinate collaborative activities (e.g., meetings). Friday's decision making on behalf of a person naturally leads to the issue of *adjustable autonomy*. An agent has the option of acting with full autonomy (e.g., delaying a meeting, volunteering the user to give a presentation, ordering a meal). On the other hand, it may act without autonomy, instead asking its user what to do. Clearly, the more decisions that Friday makes autonomously, the more time and effort it saves its user. Yet, given the high uncertainty in Friday's knowledge of its user's state and preferences, it could potentially make very costly mistakes while acting autonomously. For example, it may order an expensive dinner when the user is not hungry, or volunteer a busy user to give a presentation. Thus, each Friday must make intelligent decisions about when to consult its user and when to act autonomously.

Our initial attempt at adjustable autonomy was inspired by CAP [21], an agent system for advising a user on scheduling meetings. As with CAP, each Friday tried to learn its user preferences using decision trees under C4.5 [25]. One problem became apparent when applying this technique in Electric Elves: a user would not grant autonomy to Friday in making certain decisions, but s/he would sometimes be unavailable to provide any input at decision time. Thus, a Friday could end up waiting indefinitely for user input and miscoordinate with its teammates. We therefore modified the system so that if a user did not respond within a fixed time limit, Friday acted autonomously based on its learned decision tree. Unfortunately, when we deployed the system in our research group, it led to some dramatic failures. For instance, one user's proxy erroneously volunteered him to give a presentation. C4.5 had overgeneralized from a few examples to create an incorrect rule. Although Friday tried asking the user at first, because of the timeout, it had to eventually follow the incorrect rule and take the undesirable autonomous

action.

It was clear, based on this experience, that the team context in Electric Elves would cause difficulties for existing adjustable-autonomy techniques [4, 5, 21] that focused on solely individual human-agent interactions. Therefore, we developed a novel, decision-theoretic planning approach that used Markov Decision Processes (MDPs) [23] to support explicit reasoning about team coordination. The MDPs used in our framework [27] provide Friday with a novel three-step approach to adjustable autonomy: (i) Before transferring decision-making control, an agent explicitly weighs the cost of waiting for user input and any potential team miscoordination against the likelihood and cost of erroneous autonomous action; (ii) When transferring control, an agent does not rigidly commit to this decision, but it instead flexibly reevaluates when its user does not respond, sometimes reversing its decision and taking back autonomy; (iii) Rather than force a risky decision in situations requiring autonomous action, an agent changes its coordination arrangements by postponing or reordering activities to potentially buy time to lower decision cost/uncertainty. Since these coordination decisions and actions incur varying costs and benefits over time, agents look ahead over the different sequences of possible actions and plan a policy that maximizes team welfare.

We have implemented MDPs that model Friday's decisions on meeting rescheduling, volunteering its user to give a presentation, and selecting *which* user should give a presentation. For instance, consider one possible policy, generated from an MDP for the rescheduling of meetings. If the user has not arrived at the meeting five minutes prior to its scheduled start, this policy specifies "ask the user what to do". If the user does not arrive by the time of the meeting, the policy specifies "wait", so the agent continues acting without autonomy. However, if the user *still* has not arrived five minutes after the meeting is scheduled to start, then the policy chooses "delay by 15 minutes", which the agent then executes autonomously.

3.2 Flexible Assignment of Tasks

The human agents and software agents in our organization perform a wide variety of tasks that are often interrelated. Agents often need to delegate a subtask to another agent capable of performing it (e.g., reserve a meeting room), invoke another agent to gather and report back necessary information (e.g., find the location of a person), or rely on another agent to execute some task in the real world (e.g., attend a lunch meeting). Simple agent matchmaking is sufficient in many multi-agent systems where agents perform one (or at most a few) kind of task,

and their capabilities are designed by the system developers to fit the interactions anticipated among the agents. In contrast, our agents are complex and heterogeneous, and the agents that issue a request cannot be expected to be aware of what other agents are available and how they are invoked.

We have developed an agent matchmaker called PHOSPHORUS [8], which builds on previous research on matching problem solving goals and methods in EXPECT [28, 7]. The main features of this approach are: 1) a declarative language to express task descriptions that includes rich parameter type expressions to qualify task types; 2) task descriptions are fully translated into description logic to determine subsumption relations among tasks; 3) task descriptions are expressed in terms of domain ontologies, which provide a basis for relating and reasoning about different tasks and enables reformulation of tasks into subtasks.

Agent capabilities and requests are represented as verb clauses with typed arguments (as in a case grammar), where each argument has a name (usually a preposition) and a parameter. The type of a parameter may be a specific instance, an abstract concept (marked with spec-of), an instance type (marked with inst-of), and extensional or intensional sets of those three types. Here are some examples of capabilities of some researchers and project assistants:

```
"agents that can discuss Phosphorus"
    ((capability (discuss (obj Phosphorus-project)))
     (agents (gil surya chalupsky russ)))

"agents that can setup an LCD projector in a meeting room"
    ((capability (setup (obj (?v is (inst-of lcd-projector)))
                        (in (?r is (inst-of meeting-room)))))
     (agents (itice)))
```

Requests are formulated in the same language, and can ask about general types of instances (e.g., what agents can setup any kind of equipment for giving research presentations in a meeting room).

Description logic and subsumption reasoning are used to relate different task descriptions. Both requests and agent capabilities are translated into Loom [18]. Loom's classifier recognizes that the capability to "setup equipment" will subsume one to "setup LCD projector", because according to the domain ontologies equipment subsumes LCD projector.

PHOSPHORUS performs *task reformulations* when there are no agents with capabilities that subsume a request. In that case, it may be possible to fulfill the request by decomposing it into subtasks. This allows a more flexible matching than if one required a single agent to match all capabilities in the request. PHOSPHORUS supports set reformulations (breaking down a task on a set into its individual

elements) and covering reformulations (decomposing a task into the disjoint subclasses of its arguments). For example, no single agent can discuss the entire Electric Elves project, since no single researcher is involved in all the aspects of the project. But PHOSPHORUS can return a set of people who can collectively cover the topic based on the subprojects:

```
(COVERING -name ARIADNE-PROJECT
          -matches KNOBLOCK MINTON LERMAN
          -name PHOSPHORUS-PROJECT
          -matches GIL SURYA CHALUPSKY RUSS
          -name TEAMCORE-PROJECT
          -matches
              (COVERING
              -name ADJUSTABLE-AUTONOMY-PROJECT
                    -matches TAMBE SCERRI PYNADATH
               -name TEAMWORK-PROJECT
                    -matches TAMBE PYNADATH MODI)
          -name ROSETTA-PROJECT
          -matches GIL CHALUPSKY)
```

Many additional challenges lay ahead regarding capability representations for people within the organization. For example, although anyone has the capability to call a taxi for a visitor (and will do so if necessary), project assistants are the preferred option. Extensions to the language are needed to express additional properties of agents, such as reliability, efficiency, and invocation guidelines.

3.3 Reliable Access to Information

Timely access to up-to-date information is crucial to the successful planning and execution of tasks in the Electric Elves organization. Agents making decisions on behalf of human users need to extract information from multiple heterogeneous information sources, which include internal organizational databases (personal schedules, staff lists), and external Web sites such as airline schedules, restaurant restaurant information, traffic and weather updates, etc. For example, in order to pick a restaurant for a scheduled lunch meeting, the agents access the Restaurant Row Web site to get the locations of restaurants that meet the specified criteria, e.g., dietary restrictions. Wrappers, such as the Ariadne wrappers [13] used in the Electric Elves organization, extract data from information sources and make it available to other applications, including agents. Moreover, wrappers enable the sources, including Web sites, to be queried as if they were databases. A critical part of the wrapper is a set of extraction rules, often based on ”landmarks” or sequences of HTML tokens [22, 6, 11, 15], that enable the wrapper to quickly locate the beginning and end of data to be extracted from a page returned by the

Web source in response to some query. Previous research has primarily focused on applying machine learning techniques to rapidly generate wrappers [6, 11, 15]. Few attempts were made to provide capability to validate data, detect failures [14] and repair the wrappers when the underlying sources change in a way that breaks the wrapper. A critical component of a robust dynamic organization is the ability to automatically monitor external information sources and repair wrappers when errors are detected.

We address the problem of wrapper verification by applying machine learning techniques to learn a set of patterns that describe the information being extracted for each data field. Since the information for a single field can vary considerably, the system learns a statistical distribution of patterns. Wrappers can be verified by comparing newly extracted data to the learned patterns. When a significant difference is found, an operator can be notified or we can automatically launch the wrapper repair process.

The learned patterns represent the structure, or format, of data as a sequence of words and wildcards. Wildcards represent syntactic categories to which words belong alphabetic, numeric, capitalized, *etc*. and allow for multi-level generalization. For example, a set of street addresses all start with a pattern "*_Number_ Capitalized_*": a numeric token followed by a capitalized word. The algorithm we developed [16] finds all statistically significant starting and ending patterns in a set of positive examples of the data field. A pattern is said to be significant if it occurs more frequently than would be expected by chance if the tokens were generated randomly and independently of one another. Our approach is similar to work on grammar induction [1, 9], but our pattern language is better suited to capturing the regularities in small data fields (as opposed to languages). For the verification task, we learn the patterns from data extracted by the wrapper that is known to be correct (training examples). In the verification phase, the wrapper generates a test set of examples from pages retrieved using the same or similar set of queries. If the patterns describe statistically the same (at a given significance level) proportion of the test examples as the training examples, the wrapper is judged to be correct; otherwise, it is judged to have failed.

The wrappers frequently fail because changes in Web site format, even slight reorganizations of a page, break data extraction rules. However, since the content of the fields tends to remain the same, it is often possible to automatically repair the wrapper by learning new extraction rules. For this purpose, we exploit the patterns learned for verifying data to locate correct examples of data on new pages. For example, while the wrapper for Restaurant Row was working correctly, we managed to acquire several examples of restaurant addresses, and the verifi-

cation algorithm learned that some fraction of the examples start with the pattern "_*Number*_ *Capitalized*_" and end with the pattern "Avenue". If the Restaurant Row changes its format to look more like Zagats Web site, the wrapper will no longer extract correct addresses. The verification algorithm will detect the failure because extracted data will not be described by the learned patterns. However, since restaurant addresses will still start with the pattern "_*Number*_ *Capitalized*_" and end with the pattern "Avenue", we should be able to identify addresses on the changed pages. Once the required information has been located, these examples, along with the new pages, are provided to the wrapper generation system to learn data extraction rules for the changed site. In order to identify the correct examples of data on the changed pages we leverage both the prior knowledge about the content of data, as captured by the learned patterns, and *a priori* expectations about data. We can reasonably expect the same data field to appear in roughly the same position and in a similar context on each page; moreover, we expect at least some of the data to remain unchanged. Of course, the latter assumption is violated by information that changes on a regular basis, such as traffic and flight arrival information, though we may still rely on patterns to identify correct examples.

Our approach can be extended to automatically create wrappers for new information sources using data extracted from a known source. Thus, once we learn what restaurant addresses look like, we can use this information to extract addresses from any other yellow pages-type source, and use the extracted data to create a wrapper for the source. Such cross-site wrapper creation, as well automatic extraction of data from more complex sources, such as lists and tables, is a focus of our current research.

3.4 Knowledge from Unstructured Sources

As mentioned above, an agent-assisted organization crucially depends on access to accurate and up-to-date information about the humans it supports as well as the environment in which they operate. Some of this information can be provided directly from existing databases and online sources, but other information—people's expertise, capabilities, interests, etc.—will often not be available explicitly and might need to be modeled by hand. In a dynamic environment such as Electric Elves, however, manual modeling is only feasible for relatively static information. For example, if at some conference we want to select potential candidates for a lunch meeting with Yolanda Gil based on mutual research interests, it is not feasible to manually model relevant knowledge about each person on the conference roster before such a selection can be made.

To support team-building tasks such as inviting people for a lunch meeting, finding people potentially interested in a presentation or research meeting, finding candidates to meet with a visitor, etc., we developed a matchmaking service called the Interest Matcher. It can match people based on their research interests but also take other information into account such as involvement in research projects, present and past affiliation, universities attended, etc. To minimize the need for manual modeling in a dynamic environment, we combined statistical match techniques from the area of information retrieval (IR) with logic-based matching performed by the PowerLoom knowledge representation (KR) system. The IR techniques work well with unstructured text sources available online on the Web, which is the form in which information is typically available to outside organizations. PowerLoom facilitates declarative modeling of the decision process, modeling of missing information, logical inference, explanation and also customization.

The matchmaker's knowledge base contains an ontology of research topic areas and associated relations; rules formalizing the matchmaking process; and manually modeled, relatively static information about staff members, research projects, etc. To perform a particular matchmaking task, a requesting agent sends a message containing an appropriate PowerLoom query to the Interest Matcher. For example, the following query finds candidates for lunch with Yolanda Gil:

```
(retrieve all ?x (should-meet ?x Gil))
```

The should-meet relation and one of its supporting relations are de£ned as follows in PowerLoom:

```
(defrelation should-meet ((?p1 Person) (?p2 Person))
  :⇐ (or (interests-overlap ?p1 ?p2)
         (institution-in-common ?p1 ?p2)
         (school-in-common ?p1 ?p2)))

(defrelation interests-overlap ((?p1 Person) (?p2 Person))
  :⇐ (exists (?interest1 ?interest2)
         (and (research-interest ?p1 ?interest1)
              (research-interest ?p2 ?interest2)
              (or (subset-of ?interest1 ?interest2)
                  (subset-of ?interest2 ?interest1)))))
```

For more speci£c purposes, any of the more basic relations comprising should-meet such as interests-overlap could be queried directly by a client. Using a general purpose KR system as the matching engine provides us with this flexibility. Note, that for interests-overlap we only require a subsumption relationship, e.g., interest in planning would subsume (or overlap with) interest in hierarchical planning.

To deal with incompleteness of the KB, we allow a requesting agent to introduce new individuals and then the Interest Matcher automatically infers limited structured knowledge—their research interests—by analyzing relevant unstructured text sources on the Web.

The key idea is that people's research interests are implicitly documented in their publication record. We make these interests explicit by associating each research topic in the PowerLoom topic ontology with a statistical representation of a set of abstracts of research papers representative of the topic. These topic sets are determined automatically by querying a bibliography search engine such as Cora or the NEC ResearchIndex with seed phrases representative of the topic (access to such Web sources is facilitated by Ariadne wrappers). We then query the same search engine for publication abstracts of a particular researcher and then classify them by computing statistical similarity measures between the researcher's publications and the topic sets determined before. We use a standard IR vector space model to represent document abstracts and compute similarity by a cosine measure and by weighting terms based on how well they signify particular topic classes [26].

3.5 Coordination of Component Agents

The diverse agents in Electric Elves must work together to accomplish the complex tasks of the whole system. For instance, to plan a lunch meeting, the interest matcher finds a list of potential attendees, the Friday of each potential attendee decides whether s/he will attend, the capability matcher identifies dietary restrictions of the confirmed attendees, and the reservation site wrapper identifies possible restaurants and makes the £nal reservation. In addition to low-level communication issues, there is the complicated problem of getting all these agents to work together as a team. Each of these agents must execute its part in coordination with the others, so that it performs its tasks at the correct time and sends the results to the agents who need them.

However, constructing teams of such agents remains a difficult challenge. Current approaches to designing agent teams lack the general-purpose teamwork models that would enable agents to autonomously reason about the communication and coordination required. The absence of such teamwork models makes team construction highly labor-intensive. Human developers must provide the agents with a large number of problem-specific coordination and communication plans that are not reusable. Furthermore, the resulting teams often suffer from a lack of robustness and flexibility. In a real-world domain like Electric Elves,

teams face a variety of uncertainties, such as a member agent's unanticipated failure in fulfilling responsibilities (e.g., a presenter is delayed), members' divergent beliefs, and unexpectedly noisy communication. It is difficult to anticipate and pre-plan for all possible coordination failures.

In Electric Elves, the agents coordinate using Teamcore, a domain-independent, decentralized, teamwork-based integration architecture [24]. Teamcore uses STEAM, a general-purpose teamwork model [31] and provides core teamwork capabilities to agents by wrapping them with Teamcore proxies (separate from the Friday agents that are *user* proxies). By interfacing with Teamcore proxies, existing agents can rapidly assemble themselves into a team to solve a given problem. The Teamcore proxies form a distributed *team-readiness* layer that provides the following social capabilities: (i) coherent commitment and termination of joint goals, (ii) team reorganization in response to member failure, (iii) selective communication, (iv) incorporation of heterogeneous agents, and (v) automatic generation of tasking and monitoring requests. Although other agent-integration architectures such as OAA [20] and RETSINA [29] provide capability (iv), Teamcore's use of an explicit, domain-independent teamwork model allows it to support all five required social capabilities.

Each and every agent in the Electric Elves organization (Fridays, matchers, wrappers) has an associated Teamcore proxy that records its membership in various teams and active commitments made to these teams. Given an abstract specification of the organization and its plans, the Teamcore proxies *automatically* execute the necessary coordination tasks. They form joint commitments to team plans such as holding meetings, hosting and meeting with visitors, arranging lunch, etc. Teamcore proxies also communicate amongst themselves to ensure coherent and robust plan execution. The Teamcore proxies automatically substitute for missing roles (e.g., if the presenter is absent from the meeting) and inform each other of critical factors affecting a team plan. Finally, they communicate with their corresponding agents to monitor the agents' ability to fulfill commitments (e.g., asking Friday to monitor its user's attendance of a meeting) and to inform the agents of changes to those commitments (e.g., notifying Friday of a meeting rescheduling).

4 Electric Elves Architecture

Electric Elves is a complex and heterogeneous system spanning a wide variety of component technologies and languages, communication protocols as well as operating system platforms. Figure 2 shows the components of the current version

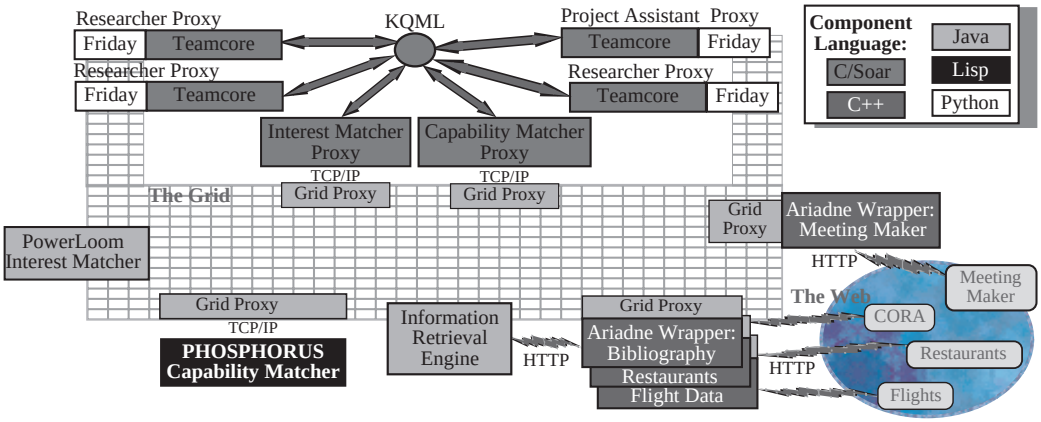


Figure 2: Electric Elves System Architecture

of Electric Elves. Teamcore agents are written in Python and Soar (which is written in C), Ariadne wrappers are written in C++, the PHOSPHORUS capability matcher is written in Common-Lisp and the PowerLoom interest matcher is written in STELLA [2] which translates into Java. The agents are distributed across SunOS 5.7, Windows NT, Windows 2000 and Linux platforms, and use TCP/IP, HTTP and the Lockheed KQML API to handle specialized communication needs.

Tying all these different pieces together in a robust and coherent manner constitutes a significant engineering challenge. Initially we looked for an implementation of KQML, but there was none available that supported all the languages and platforms we required. To solve this integration problem, we are using the DARPA supported CoABS Grid technology developed by Global InfoTek, Inc. and ISX Corporation[1]. The CoABS Grid is a Java-based communication infrastructure built on top of Sun's Jini networking technology. It provides message and service-based communication mechanisms, agent registration, lookup and discovery services, as well as message logging, security and visualization facilities. Since it is is written in Java, it runs on a wide variety of OS platforms, and it is also relatively easy to connect with non-Java technology. Grid proxy components connect non-Java technology to the Grid.

We primarily use the CoABS Grid as a uniform transport mechanism. The content of Grid messages are in KQML format and could potentially be communicated via alternative means. Not all Electric Elves message traffic goes across the Grid. For example, the Teamcore agents communicate via their own protocol (the

[1] http://coabs.globalinfotek.com/coabs_public/coabs_pdf/gridvision.pdf

Lockheed KQML API) and only use the Grid to communicate with non-Teamcore agents such as the capability and interest matchers. Similarly, the information retrieval engine communicates with Ariadne wrappers directly via HTTP instead of going through the Grid.

5 Related Work

Several agent-based systems have been developed that support specific tasks within an organization, such as meeting scheduling [3] and visitor hosting [12, 30]. In contrast to these systems, we believe that our approach integrates a range of technologies that can support a variety of tasks within the organization. Agent architectures have been applied to organizational tasks [29, 20, 17], but none of them include technology for team work, adjustable autonomy, and dynamic collection of information from external sources.

To our knowledge, Electric Elves represents the first agent-based system that is used for routine tasks within a human organization. Several other areas of research have looked at complementary aspects of the problems that we aim to address. Research on architectures and systems for Computer-Supported Cooperative Work include a variety of information management and communication technologies that facilitate collaboration within human organizations [10, 19]. In contrast with our work, they do not have agents associated with people that have some degree of autonomy and can make decisions on a human's behalf. Our work is also complementary and can be extended with ongoing research on ubiquitous computing and intelligent buildings [17]. These projects are embedding sensor networks and agents to control and improve our everyday physical environments. This kind of infrastructure would make it easier for Electric Elves to locate and contact people as well as to direct the environmental control agents in support of organizational tasks.

6 Current Status

The Electric Elves system has been in use within our research group at ISI since June 1, 2000; and operating continuously 24 hours a day, 7 days a week (with interruptions for bug fixes and enhancements). Usually, nine agent proxies are working for nine users, with one proxy each for a capability matcher and an interest matcher. The proxies communicate with their users using a variety of devices:

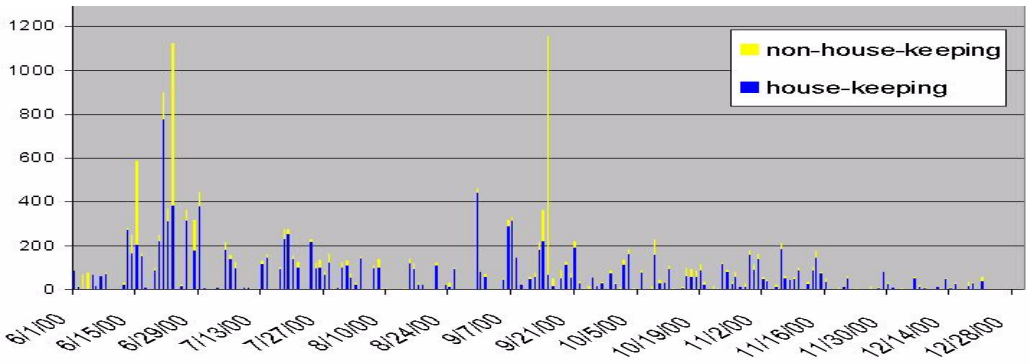


Figure 3: Number of daily coordination messages exchanged by proxies over a seven-month period.

workstation display, voice, mobile phones, and palm pilots. They also communicate with restaurants by sending faxes.

Figure 3 plots the number of daily messages exchanged by the proxies for seven months (June 1, 2000 to December 31, 2000). The size of the daily counts demonstrates the large amount of coordination actions necessary in managing all of the activities such as meeting rescheduling. The high variability is due to the variance in the number of daily activities, e.g., weekends and long breaks such as the Christmas break, usually have very little activity. Furthermore, with continually increasing system stability, the amount of housekeeping activity necessary has reduced automatically.

Several observations show the effectiveness of Electric Elves. First, over the past several months, few emails have been exchanged among our group members indicating to each other that they may get delayed to meetings. Instead, Friday agents automatically address such delays. Also the overhead of waiting for delayed members in meeting rooms has been reduced. Overall, 1128 meetings have been monitored, out of which 285 have been rescheduled, 230 automatically and 55 by hand. Both autonomous rescheduling and human intervention were useful in Elves.

Furthermore, whereas in the past, one of our group members would need to circulate emails trying to recruit a presenter for research meetings and making announcements, this overhead has almost completely vanished—weekly auctions automatically select the presenters at our research meetings. These auctions are automatically opened when the system receives notification of any meeting requiring a presentation. Auction decisions may be made without requiring a full set of

bids; in fact, in one case, only 4 out of 9 possible bids were received. The rest of the group simply did not bid before the winner was announced. Most of the time, the winner was automatically selected. However, on two occasions (July 6 and Sept 19) exceptional circumstances (e.g., a visitor) required human intervention, which our proxy team easily accommodates.

7 Discussion

As described in this paper we have successfully deployed the Electric Elves in our own real-world organization. These agents interact directly with humans both within the organization and outside the organization communicating by email, wireless messaging, and faxes. Our agents go beyond simply automating tasks that were previously performed by humans. Because hardware and processing power is cheap, our agents can perform a level of monitoring that would be impractical for human assistants, ensuring that activities within an organization run smoothly and that events are planned and coordinated to maximize the productivity of the individuals of an organization.

In the process of building the applications described in this paper we addressed an number of key technology problems that arise in any agent-based system applied to human organizations. In particular we described how to use Markov Decision Processes to determine the appropriate degree of autonomy for the agents, how to use knowledged-based matchmaking to assign tasks within an organization, how to apply machine learning techniques to ensure robust access to the data sources, how to combine knowledge-based and statistical matchmaking techniques to derive knowledge about the participants both within and outside an organization, and how to apply multi-agent teamwork coordination to dynamically assemble teams.

Acknowledgements

The research reported here was supported in part by the Rome Laboratory of the Air Force Systems Command and the Defense Advanced Research Projects Agency (DARPA) under contract numbers F30602-97-C-0068, F30602-98-2-0109 and F30602-98-2-0108; and in part by the Air Force Office of Scientific Research under Grant Number F49620-01-1-0053. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

Surya Ramachandran also contributed to the success of this project.

References

[1] R. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In *Second International Colloquium on Grammatical Inference and Applications*, number 862 in Lecture Notes In Computer Science, pages 139–152. Springer Verlag, Berlin, 1994.

[2] H. Chalupsky and R.M. MacGregor. STELLA – a Lisp-like language for symbolic programming with delivery in Common Lisp, C++ and Java. In *Proc. of the 1999 Lisp User Group Meeting*, Berkeley, CA, 1999. Franz Inc.

[3] Lisa Dent, Jesus Boticario, John McDermott, Tom Mitchell, and David Zabowski. A personal learning apprentice. In *Proc. of AAAI-1992*, pages 96–103, San Jose, CA., 1992.

[4] Gregory A. Dorais, R. Peter Bonasso, David Kortenkamp, Barney Pell, and Debra Schreckenghost. Adjustable autonomy for human-centered autonomous systems on Mars. In *Proceedings of the First International Conference of the Mars Society*, pages 397–420, 1998. Boulder, Colorado August 13–16, 1998.

[5] G. Ferguson, J. Allen, and B. Miller. TRAINS-95: Towards a mixed initiative planning assistant. In *Proceedings of the Third Conference on Arti£cial Intelligence Planning Systems*, pages 70–77, May 1996. Edinburgh, Scotland May 29–31, 1996.

[6] D. Freitag and N. Kushmerick. Boosted wrapper induction. In *Proc. of AAAI-2000*.

[7] Y. Gil and P. Gonzalez. Subsumption-based matching: Bringing semantics to goals. In *International Workshop on Description Logics*, 1996. Boston, MA November 2–4, 1996.

[8] Y. Gil and S. Ramachandran. PHOSPHORUS: A task-based agent matchmaker. In *Proceedins of the Fifth International Conference on Autonomous Agents*, May 2001. Montreal, Canada May 28 – June 1, 2001.

[9] T. Goan, N. Benson, and O. Etzioni. A grammar inference algorithm for the world wide web. In *Proc. of the AAAI Spring Symposium on Machine Learning in Information Access*, 1996.

[10] Saul Greenberg, editor. *Computer Supported Cooperative Work and Groupware*. Academic Press, London, 1991.

[11] C. Hsu and M. Dung. Generating £nite-state transducers for semi-structured data extraction from the web. *Journal of Information Systems*, 23(8):521–538, 1998.

[12] Henry Kautz, Bart Selman, Michael Coen, and Steven Ketchpel. An experiment in the design of software agents. In *Proc. of AAAI-1994*, 1994.

[13] Craig A. Knoblock, Steven Minton, Jose Luis Ambite, Pragnesh Jay Modi Naveen Ashish, Ion Muslea, Andrew G. Philpot, and Sheila Tejada. Modeling web sources for information integration. In *Proc. of AAAI-1998*, 1998.

[14] N. Kushmerick. Regression testing for wrapper maintenance. In *Proc. of AAAI-1999*, pages 74–79, 1999.

[15] N. Kushmerick. Wrapper induction: ef£ciency and expressiveness. *Arti£cial Intelligence*, 118(1-2):15–68, 2000.

[16] K. Lerman and S. Minton. Learning the common structure of data. In *Proc. of AAAI-2000*, pages 609–614, 2000.

[17] Victor Lesser, Michael Atighetchi, Brett Benyo, Bryan Horling, Anita Raja, Rgis Vincent, Thomas Wagner, Ping Xuan, and Shelley XQ. Zhang. The UMASS intelligent home project. In *Proceedings of the Third Annual Conference on Autonomous Agents*, pages 291–298. ACM Press, New York, 1999.

[18] R.M. MacGregor. Inside the LOOM description classifier. *ACM SIGART Bulletin*, 2(3):88–92, 1991.

[19] Thomas W. Malone, Kevin Crowston, Jintae Lee, Brian Pentland, Chrysanthos Dellarocas, George Wyner, John Quimby, Charley Osborne, and Abraham Bernstein. *Tools for inventing organizations: Toward a handbook of organizational processes*. Center for Coordination Science, Massachusetts

Institute of Technology, Cambridge, Massachusetts, 1997. Working Paper No. 198.

[20] David L. Martin, Adam J. Cheyer, and Douglas B. Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1-2):92–128, 1999.

[21] Tom Mitchell, Rich Caruana, Dayne Freitag, John McDermott, and David Zabowski. Experience with a learning personal assistant. *Communications of the ACM*, 37(7):81–91, July 1994.

[22] I. Muslea, S. Minton, and C. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Journal of Autonomous Agents and Multi-Agent Systems*, 4(1/2), 2000.

[23] M. L. Puterman. *Markov Decision Processes*. John Wiley & Sons, 1994.

[24] David V. Pynadath, Milind Tambe, Nicolas Chauvat, and Lawrence Cavedon. Toward team-oriented programming. In Nicholas R. Jennings and Yves Lespérance, editors, *Intelligent Agents VI: Agent Theories, Architectures and Languages*, pages 233–247. Springer-Verlag, 1999.

[25] J. R. Quinlan. *C4.5: Programs for machine learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[26] G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Tokio, 1983.

[27] Paul Scerri, David V. Pynadath, and Milind Tambe. Adjustable autonomy in real-world multi-agent environments. In *Proceedings of the Fifth Conference on Autonomous Agents*, pages 300–307. ACM Press, New York, 2001.

[28] W. R. Swartout and Y. Gil. EXPECT: Explicit representations for ¤exible acquisition. In *Proceedings of the Ninth Knowledge Acquisition for Knowledge-Based Systems Workshop*, 1995. Banff, Canada February 26 – March 3, 1995.

[29] K. Sycara, K. Decker, A. Pannu, M. Williamson, and D. Zeng. Distributed intelligent agents. *IEEE Expert*, 1996.

[30] Katia Sycara and Dajun Zeng. Towards an intelligent electronic secretary. In *CIKM-94 (International Conference on Information and Knowledge Management), Intelligent Information Agents Workshop*, Gaithersburg, MD, 1994.

[31] Milind Tambe. Towards ¤exible teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1997.

Appendix B:

Getting from Here to There: Interactive Planning and Agent Execution for Optimizing Travel*†

José Luis Ambite, Greg Barish,
Craig A. Knoblock, Maria Muslea, Jean Oh
Information Sciences Institute
University of Southern California
4676 Admiralty Way, Marina del Rey, CA 90292, USA
{ambite,barish,knoblock,mariam,jeanoh}@isi.edu

Steven Minton
Fetch Technologies
4676 Admiralty Way,
Marina del Rey, CA 90292, USA
steve.minton@fetch.com

Abstract

Planning and monitoring a trip is a common but complicated human activity. Creating an itinerary is nontrivial because it requires coordination with existing schedules and making a variety of interdependent choices. Once planned, there are many possible events that can affect the plan, such as schedule changes or flight cancellations, and checking for these possible events requires time and effort. In this paper, we describe how Heracles and Theseus, two information gathering and monitoring tools that we built, can be used to simplify this process. Heracles is a hierarchical constraint planner that aids in interactive itinerary development by showing how a particular choice (e.g., destination airport) affects other choices (e.g., possible modes of transportation, available airlines, etc.). Heracles builds on an information agent platform, called Theseus, that provides the technology for efficiently executing agents for information gathering and monitoring tasks. In this paper we present the technologies underlying these systems and describe how they are applied to build a state-of-the-art travel system.

*Appeared in *Proceedings of the Fourteenth Conference on Innovative Applications of Artificial Intelligence (IAAI-2002)*, pages 862–869, Edmonton, Alberta, Canada, 2002.

1 Introduction

The standard approach to planning business trips is to select the flights, reserve a hotel, and possibly a car at the destination. Choices of which airports to fly into and out of, whether to park at the airport or take a taxi, and whether to rent a car at the destination are often ad hoc choices based on past experience. These choices are frequently suboptimal, but the time and effort required to make more informed choices usually outweighs the cost. Similarly, once a trip has been planned it is usually ignored until a few hours before the first flight. A traveler might check on the status of the flights or use one of the services that automatically notify a traveler of flight status information, but otherwise a traveler just copes with problems that arise as they arise. Beyond flight delays and cancellations there a variety of possible events that occur in the real world that one would ideally like to anticipate, but again the cost and effort required to monitor for these events is not usually deemed to be worth the trouble. Schedules can change, prices may go down after purchasing a ticket, flight delays can result in missed connections, and hotel rooms and rental cars are given away because travelers arrive late.

To address these issues we have developed an integrated travel planning and monitoring system. The *Travel Assistant* provides an interactive approach to making travel plans where all of the information required to make informed choices is available to the user. For example, when deciding whether to park at the airport or take a taxi, the system compares the cost of parking and the cost of a taxi given other selections, such as the airport, the specific parking lot, and the starting location of the traveler. Likewise, when the user is deciding which airport to fly into, the system provides not only the cost of the flights, but also determines the impact on the cost of the ground transportation at the destination. Once a trip is planned, the monitoring tasks are addressed by a set of information agents that can attend to details for which it would be impractical for a human assistant to monitor. For example, beyond simply notifying a traveler of flight delays, an agent will also send faxes to the hotel and car rental agencies to notify them of the delay and ensure that the room and car will be available. Likewise, when a traveler arrives in a city for a connecting flight, an agent notifies the traveler if there are any earlier connecting flights and provides both arrival and departure gates.

These innovations in travel planning and monitoring are made possible by two underlying AI technologies. The first is the Heracles interactive constraint-based planner [Knoblock *et al.*, 2001], which captures the interrelationships of the data and user choices using a set of constraint rules. Using Heracles we can easily define a system for interactively planning a trip. The second is the Theseus information agent platform [Barish *et al.*, 2000], which facilitates the rapid creation of information gathering and monitoring agents. These agents provide data to the Heracles planner and keep track of information changes relevant to the travel plans. Based on these technologies, we have developed a complete end-to-end travel planning and monitoring system that is in use today.

The remainder of this paper describes the travel application and underlying technology in more detail. The next section describes by example the trip planning process as well as the monitoring agents that ensure that the trip is executed smoothly. Then, we present the constraint-based planning technology that supports the trip planning. Next, we describe the

information agent technology, which provides the data for the planner and the agents for monitoring the trip. Finally, we compare with related work, and discuss our contributions and future plans.

2 Planning and Monitoring a Trip

In this section we describe by example the functionality and interface of our *Travel Assistant*, showing both its capabilities for interactive planning and for monitoring the travel plans.

2.1 Interactive Travel Planning

Our *Travel Assistant* starts the travel planning process by retrieving the business meetings from the user's calendar program (e.g., Microsoft Outlook). Figure 1 shows the user interface of the *Travel Assistant* with the high level information for planning a trip to attend a business meeting. The interface displays a set of boxes showing values, which we call *slots*. A slot holds a current value and a set of possible values, which can be viewed in a pull-down list by clicking the arrow at the right edge of the slot. For example, there are slots for the subject and location of the meeting with values Travel Planner Meeting and DC respectively. The user could choose to plan another meeting from the list or input meeting information directly.

Once the system has the details of the meeting, the next step is to determine how to get to the destination. There are three possible modes of transportation: Fly, Drive, or Take a Taxi. The system recommends the transportation mode based on the distance between the user's location and the meeting location. The system obtains the departure location from the user's personal profile and the meeting location from Outlook. The system computes the distance by first geocoding (determining the latitude and longitude) of the origin and destination addresses using the Mapblast Web site (www.mapblast.com). Then, using the geocoded information, a local constraint computes the distance between the two points. In our example, the distance between Los Angeles and Washington D.C. is 2,294 miles, so the system recommends that the user Fly. Since the meeting lasts for several days, it also recommends that the user stay at a hotel. Of course, the user can always override the system suggestions.

The *Travel Assistant* organizes the process of trip planning and the associated information hierarchically. The left pane of Figure 1 shows this hierarchical structure, with the particular choices made for the current plan. In this example, the trip consists of three tasks: flying to the meeting, staying at a hotel at the meeting location, and flying back home. Some tasks are further divided into subtasks, for example, how to get to and from the airport when flying. In Heracles these tasks are represented by related slots and constraints that are encapsulated into units called *templates*, which are organized hierarchically.

The *Travel Assistant* helps the user evaluate tradeoffs that involve many different pieces of information and calculations. For example, Figure 2 illustrates how the system recommends the mode of transportation to the departure airport. This recommendation is made by comparing the cost of parking a car at the airport for the duration of the trip to the cost of taking a taxi to and from the airport. The system computes the cost of parking by retrieving the available airport parking lots and their daily rates from the AirWise

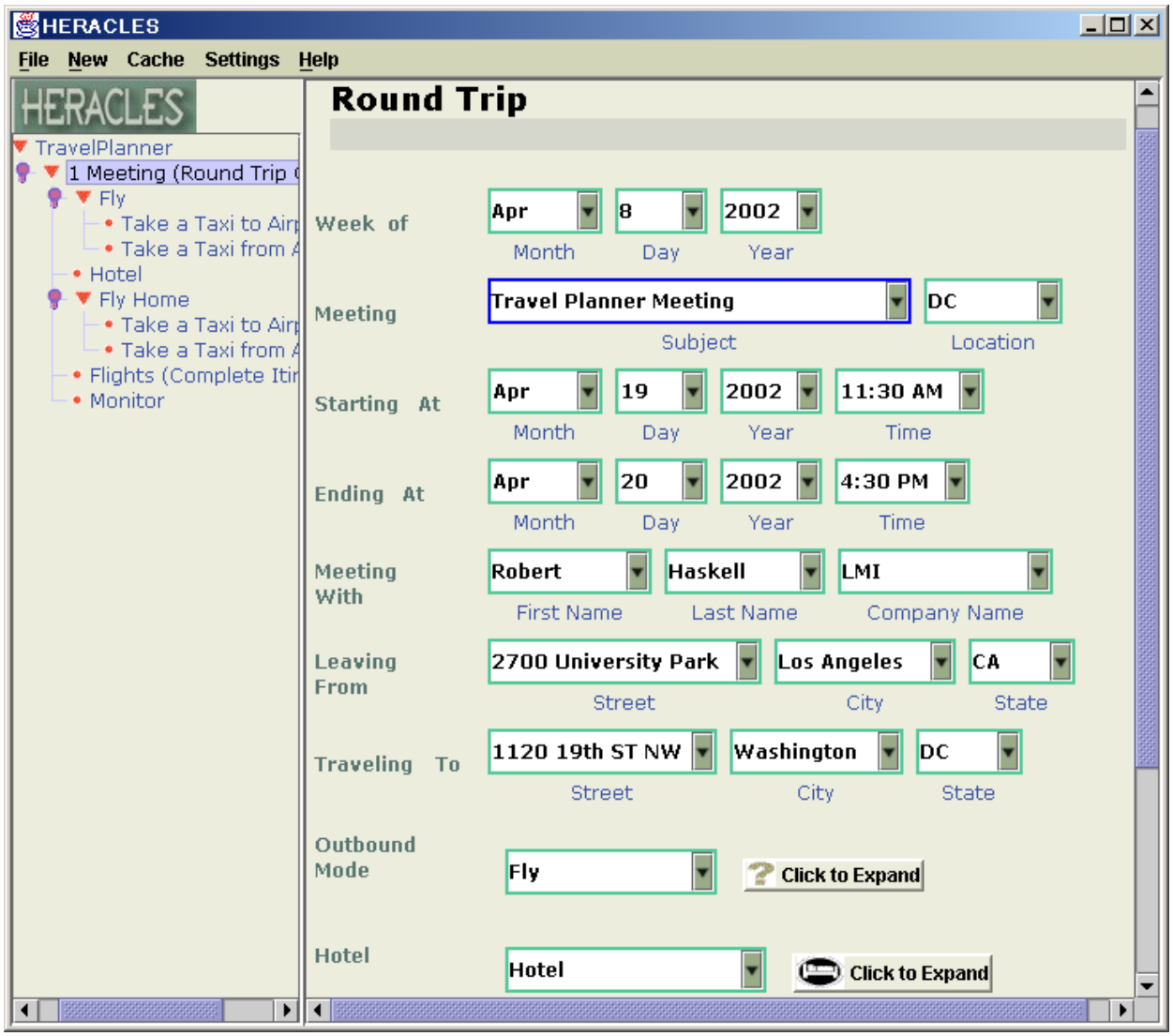


Figure 1: Planning a Meeting

site (www.airwise.com), determining the number of days the car will be parked based on scheduled meetings, and then calculating the total cost of parking. Similarly, the system computes the taxi fare by retrieving the distance between the user's home address and the departure airport from the Yahoo Map site (maps.yahoo.com), retrieving the taxi fare from the Washington Post site (www.washingtonpost.com), and then calculating the total cost. Initially, the system recommends taking a taxi since the taxi fare is only $19.50, while the cost of parking would be $48.00 using the Terminal Parking lot (the preferred parking lot in the user's profile). However, when the user changes the selected parking lot to Economy Lot B, which is $5 per day, this makes the total parking rate cheaper than the taxi fare, so the system changes the recommendation to Drive.

The system actively maintains the dependencies among slots so that changes to earlier decisions are propagated throughout the travel planning process. For example, Figure 3

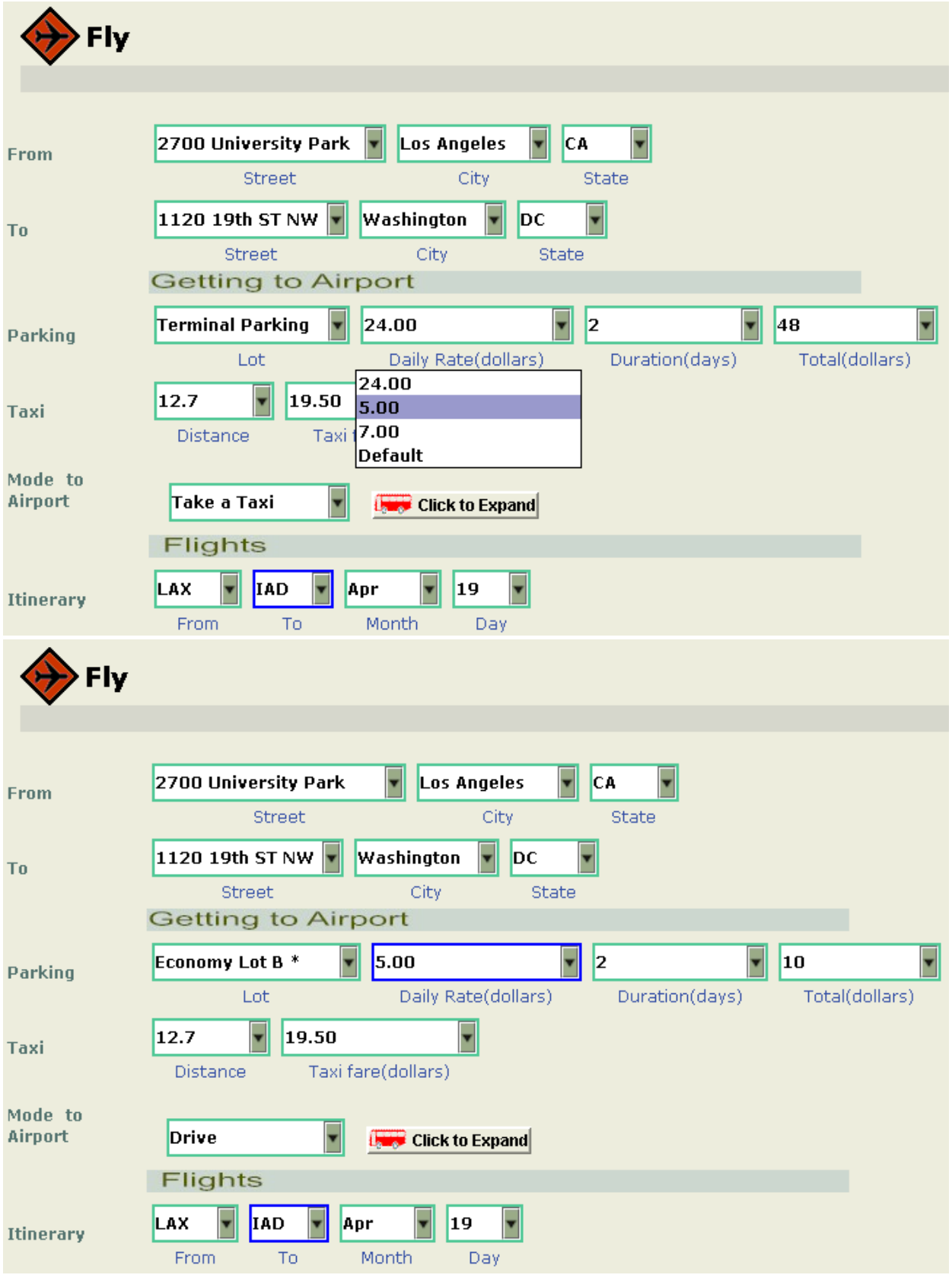


Figure 2: Comparing Cost of Driving versus Taking a Taxi

shows how the Taxi template is affected when the user changes the departure airport in the higher-level Round Trip Flights template. In the original plan, the flight departs from Los Angeles International (LAX) at 11:55 PM. The user's preference is to arrive an hour before the departure time, thus he/she needs to arrive at LAX by 10:55 PM. Since Mapblast calculates that it takes 24 minutes to drive from the user's home to LAX, the system recommends leaving by 10:31 PM. When the user changes the departure airport from LAX to Long Beach airport (LGB), the system retrieves a new map and recomputes the driving time. Changing the departure airport also results in a different set of flights. The recommended flight from LGB departs at 9:50 PM and driving to LGB takes 28 minutes. Thus, to arrive at LGB by 8:50 PM, the system now suggests leaving home by 8:22 PM.

2.2 Monitoring Travel Status

There are various dynamic events that can affect a travel plan, for instance, flight delays, cancellations, fare reductions, etc. Many of these events can be detected in advance by monitoring information sources. The *Travel Assistant* is aware that some of the information it accesses is subject to change, so it delegates the task of following the evolution of such information to a set of monitoring agents. For instance, a flight schedule change is a critical event since it can have an effect not only on the user's schedule at the destination but also on the reservations at a hotel and a car rental agency. In addition to agents handling critical events, there are also monitoring agents whose purpose is to make a trip more convenient or cost-effective. For example, tracking airfares or finding restaurants near the current location of the user. In what follows, we describe the monitoring agents that we defined for travel planning. As shown in Figure 4, Heracles automatically generates the set of agents for monitoring a travel plan. Figure 5 shows some example messages sent by these agents.

The *Flight-Status* monitoring agent uses the ITN Flight Tracker site to retrieve the current status of a given flight. If the flight is on time, the agent sends the user a message to that effect two hours before departure. If the user's flight is delayed or canceled, it sends an alert through the user's preferred device (e.g., a text message to a cellular phone). If the flight is delayed by more than an hour, the agent sends a fax to the car rental counter to confirm the user's reservation. If the flight is going to arrive at the destination airport after 5 PM, the agent sends another fax to the hotel so that the reserved room will not be given away. Since the probability of a change in the status of a flight is greater as the departure time gets closer, the agent monitors the status more frequently as the departure time nears.

The *Airfare* monitoring agent keeps track of current prices for the user's itinerary. The airlines change prices unpredictably, but the traveler can request a refund (for a fee) if the price drops below the purchase price. This agent gathers current fares from Orbitz (www.orbitz.com) and notifies the user if the price drops by more than a given threshold.

The *Flight-Schedule* monitoring agent keeps track of changes to the scheduled departure time of a flight (using Orbitz) and notifies the user if there is any change. Without such an agent, a traveler often only discovers this type of schedule changes after arriving at the airport.

The *Earlier-Flight* monitoring agent uses Orbitz to find the flights that leave earlier than the scheduled flight. It shows the alternative earlier flights and their status. This information

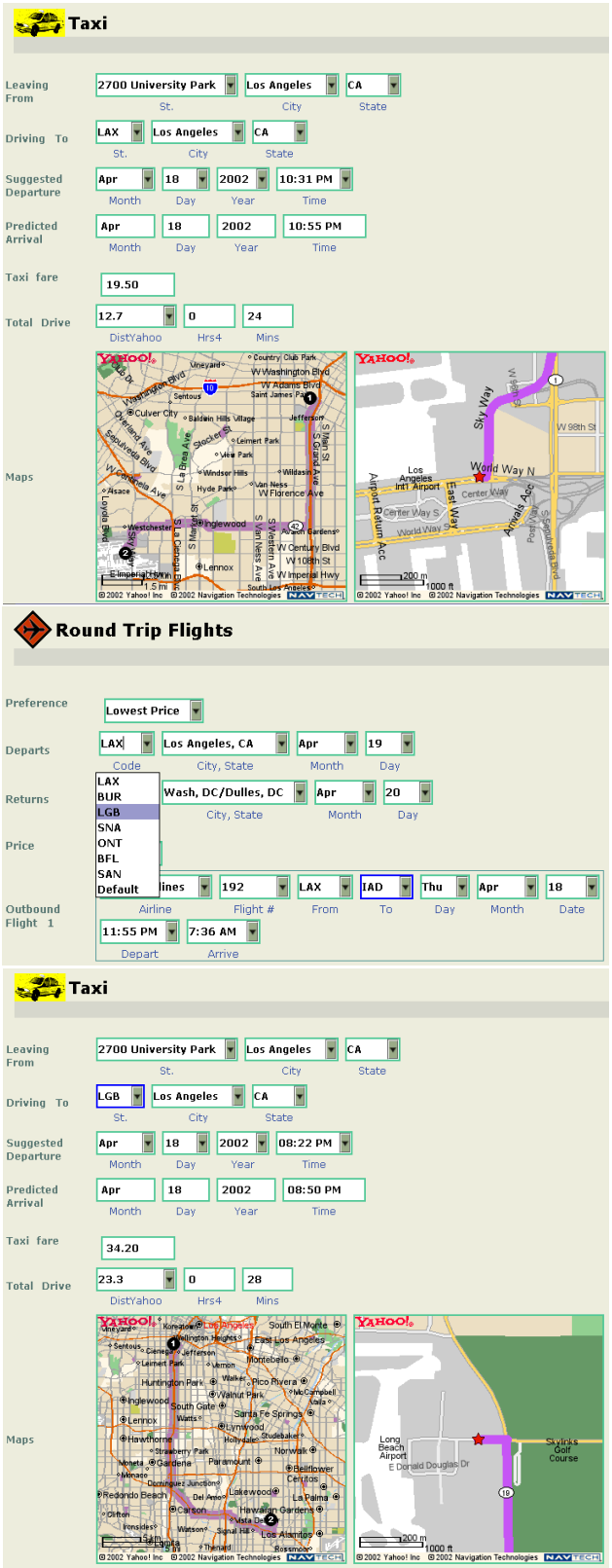


Figure 3: Change in Selected Airport Propagates to Drive Subtemplate

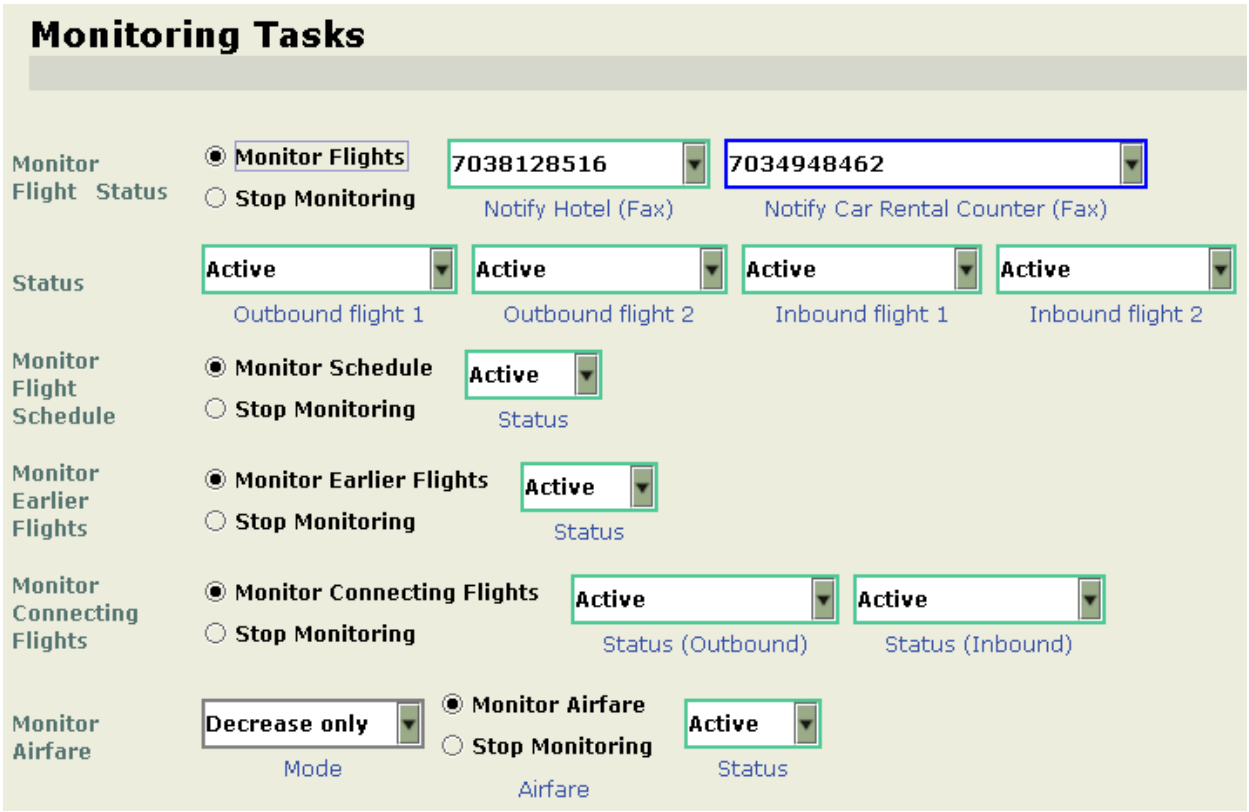


Figure 4: Template for Generating Monitoring Agents

(a) *Flight-Status Agent*: Flight delayed message

Your United Airlines flight 190 has been delayed. It was originally scheduled to depart at 11:45 AM and is now scheduled to depart at 12:30 PM. The new arrival time is 7:59 PM.

(b) *Flight-Status Agent*: Flight cancelled message

Your Delta Air Lines flight 200 has been cancelled.

(c) *Flight-Status Agent*: Fax to a hotel message

Attention : Registration Desk

I am sending this message on behalf of David Pynadath, who has a reservation at your hotel. David Pynadath is on United Airlines 190, which is now scheduled to arrive at IAD at 7:59 PM. Since the flight will be arriving late, I would like to request that you indicate this in the reservation so that the room is not given away.

(d) *Airfare Agent*: Airfare dropped message

The airfare for your American Airlines itinerary (IAD - LAX) dropped to $281.

(e) *Earlier-Flight Agent*: Earlier flights message

The status of your currently scheduled flight is:

190 LAX (11:45 AM) - IAD (7:29 PM) 45 minutes Late

The following United Airlines flight arrives earlier than your flight:

946 LAX (8:31 AM) - IAD (3:35 PM) 11 minutes Late

Figure 5: Actual Messages sent by Monitoring Agents

becomes extremely useful when the scheduled flight is significantly delayed or canceled.

The *Flight-Connection* agent tracks the user's current flight, and a few minutes before it lands, it sends the user the gate and status of the connecting flight.

The *Restaurant-Finder* agent locates the user based on either a Global Positioning System (GPS) device or his/her expected location according to the plan. On request, it suggests the five closest restaurants providing cuisine type, price, address, phone number, latitude, longitude, and distance from the user's location.

In the following sections, we describe in detail the technology we have used to automate travel planning and monitoring, namely, the Heracles interactive constraint-based planning framework and the Theseus agent execution and monitoring system.

3 Interactive Constraint-based Planning

The critical challenge for Heracles is integrating multiple information sources, programs, and constraints into a cohesive, effective tool. We saw examples of these diverse capabilities in the *Travel Assistant*, such as retrieving scheduling information from a calendar system, computing the duration of a given meeting, and invoking an information agent to find directions to the meeting.

Constraint reasoning technology offers a clean way to integrate multiple heterogeneous subsystems in a plug-and-play approach. Our approach employs a constraint representation where we model each piece of information as a distinct variable[1] and describe the relations that define the valid values of a set of variables as constraints. A constraint can be implemented either by a local procedure within the constraint engine or by an external component. In particular, we use information agents built with Theseus to implement many of the external constraints.

Using a constraint-based representation as the basis for control has the advantage that it is a declarative representation and can have many alternative execution paths. Thus, we need not commit to a specific order for executing components or propagating information. The constraint propagation system will determine the execution order in a natural manner. The constraint reasoning system propagates information entered by the user as well as the system's suggestions, decides when to launch information requests, evaluates constraints, and computes preferences.

3.1 Constraint Network Representation

A constraint network is a set of variables and constraints that interrelate and define the valid values for the variables. Figure 6 shows the fragment of the constraint network of the *Travel Assistant* that addresses the selection of the method of travel from the user's initial location to the airport. The choices under consideration are: driving one's car (which implies leaving it parked at the airport for the duration of the trip) or taking a taxi.

[1]In the example in the previous section we have referred to each piece of information presented to the user as a slot. We use the term slot for user interface purposes. Each slot has a corresponding variable defined in the constraint network, but there may be variables that are not presented to the user.

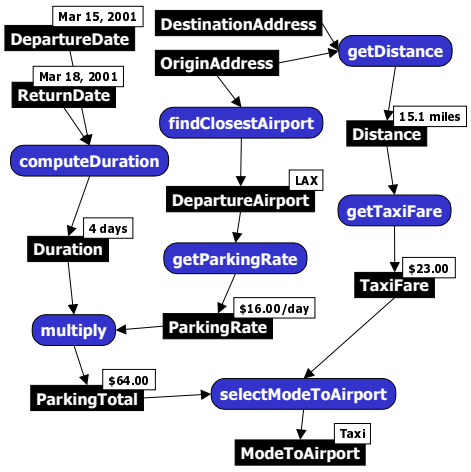


Figure 6: Constraint Network: Driving Versus Taking a Taxi

In the sample network of Figure 6, the variables are shown as dark rectangles and the assigned values as white rectangles next to them. The variables capture the relevant information for this task in the application domain, such as the DepartureDate, the Duration of the trip, the ParkingTotal (the total cost of parking for the duration of the trip), the TaxiFare, and the ModeToAirport. The DepartureAirport has an assigned value of LAX, which is assigned by the system since it is the closest airport to the user's address.

Conceptually, a *constraint* is a n-ary predicate that relates a set of n variables by defining the valid combinations of values for those variables. A constraint is a computable component which may be implemented by a local table look-up, by the computation of a local function, by retrieving a set of tuples from a remote information agent, or by calling an arbitrary external program. In Heracles the constraints are directed. The system evaluates a constraint only after it has assigned values to a subset of its variables, the *input* variables. The constraint evaluation produces the set of *possible values* for the *output* variables.

In the sample network of Figure 6 the constraints are shown as rounded rectangles. For example, the computeDuration constraint involves three variables (DepartureDate, ReturnDate, and Duration), and it's implemented by a function that computes the duration of a trip given the departure and return dates. The constraint getParkingRate is implemented by calling an information agent that accesses a web site that contains parking rates for airports in the US.

Each variable can also be associated with a preference constraint. Evaluating the preference constraint over the possible values produces the *assigned* value of the variable. (The user can manually reset the assigned value at any point by selecting a different alternative.) Preference constraints are often implemented as functions that impose an ordering on the values of a domain. An example of a preference for business travel is to choose a hotel closest to the meeting.

3.2 Constraint Propagation

Since the constraints are directed, the constraint network can be seen as a directed graph. In the current version of the system, this constraint graph must be acyclic, which means that information flows in one direction. This directionality simplifies the interaction with the user. Note that if the user changes a variable's value, this change may need to be propagated throughout the constraint graph.

The constraint propagation algorithm proceeds as follows. When a given variable is assigned a value, either directly by the user or by the system, the algorithm recomputes the possible value sets and assigned values of all its dependent variables. This process continues recursively until there are no more changes in the network. More specifically, when a variable X changes its value, the constraints that have X as input variable are recomputed. This may generate a new set of possible values for each dependent variable Y. If this set changes, the preference constraint for Y is evaluated selecting one of the possible values as the new assigned value for Y. If this assigned value is different from the previous one, it causes the system to recompute the values for further downstream variables. Values that have been assigned by the user are always preferred as long as they are consistent with the constraints.

Consider again the sample constraint network in Figure 6. First, the constraint that finds the closest airport to the user's home address assigns the value LAX to the variable DepartureAirport. Then, the constraint getParkingRate, which is a call to an information agent, fires producing a set of rates for different parking lots. The preference constraint selects terminal parking which is $16.00/day. This value is multiplied by the duration of the trip to compute the ParkingTotal of $64 (using the simple local constraint multiply). A similar chain of events results in the computation of the TaxiFare. Once the ParkingTotal and the TaxiFare are computed, the selectModeToAirport constraint compares the costs and chooses the cheapest means of transportation, which in this case is to take a Taxi.

3.3 Template Representation

As described previously, to modularize an application and deal with its complexity, the user interface presents the planning application as a hierarchy of templates. For example, the top-level template of the *Travel Assistant* (shown in Figure 1) includes a set of slots associated with who you are meeting with, when the meeting will occur, and where the meeting will be held. The templates are organized hierarchically so that a higher-level template representing an abstract task (e.g., Trip) may be decomposed into a set of more specific subtasks, called subtemplates (e.g., Fly, Drive, etc). This hierarchical task network structure helps to manage the complexity of the application for the user by hiding less important details until the major decisions have been achieved.

This template-oriented organization has ramifications for the constraint network. The network is effectively divided into partitions, where each partition consists of the variables and constraints that compose a single template. During the planning process the system only propagates changes to variables within the template that the user is currently working on. This strategy considerably improves performance.

4 Information Agents

Our system uses information agents to support both the trip planning and monitoring. While information agents are similar to other types of software agents, their plans are distinguished by a focus on gathering, integrating, and monitoring of data from distributed and remote sources. To efficiently perform these tasks we use Theseus [Barish *et al.*, 2000, Barish & Knoblock, 2002], which is a streaming dataflow architecture for plan execution. In this section, we describe how we use Theseus to build agents capable of efficiently gathering and monitoring information from remote data sources.

4.1 Defining an Information Agent

Building an information agent requires defining a plan in Theseus. Each plan consists of a name, a set of input and output variables, and a network of operators connected in a producer-consumer fashion. For example, the *Flight-Status* agent takes flight data (i.e., airline, flight number) as input and produces status information (i.e., projected arrival time) as output.

Each operator in a plan receives input data, processes it in some manner, and outputs the result - which is then potentially used by another operator. Operators logically process and transmit data in terms of relations, which are composed of a set of tuples. Each tuple consists of a set of attributes, where an attribute of a relation can be either a string, number, nested relation, or XML object.

The set of operators in Theseus support a range of capabilities. First, there are information gathering operators, which retrieve data from local and remote sources including web sites. Second, there are data manipulation operators, which provide the standard relational operations, such as select, project and join, as well as XML manipulation operations using *XQuery*. Third, there are monitoring-related operators, which provide scheduling and unscheduling of tasks and communication with a user through email or fax.

Plans in Theseus are just like operators: they are named and have input and output arguments. Consequently, any plan can be called as an operator from within any other plan. This subplan capability allows a developer to define new agents by composing existing ones.

4.2 Accessing Web Sources

Access to on-line data sources is a critical component of our information agents. In the *Travel Assistant* there is no data stored locally in the system. Instead, all information is accessed directly from web sources. To do this we build *wrappers* that enable web sources to be queried as structured data sources. This makes it easy for the system to manipulate the resulting data as well as integrate it with information from other data sources.

For example, a wrapper for Yahoo Weather dynamically turns the source into XML data in response to a query. Since the weather data changes frequently, it would not be practical to download this data in advance and cache it for future queries. Instead, the wrapper provides access to the live data, but provides it in a structured form.

We have developed a set of tools for semi-automatically creating wrappers for web sources

[Knoblock *et al.*, 2000]. The tools allow a user to specify by example what the wrapper should extract from a source. The examples are then fed to an inductive learning system that generates a set of rules for extracting the required data from a site.

Once a wrapper for a site has been created, Theseus agents can programmatically access data from that site using the wrapper operator in their plans. For example, with the wrapper for Yahoo Weather, we can now send a request to get the weather for a particular city and it will return the corresponding data.

4.3 Monitoring Sources

In addition to being able to gather data from web sources, Theseus agents are capable of *monitoring* those sources and performing a set of actions based on observed changes. The monitoring is performed by retrieving data from online sources and comparing the returned results against information that was previously retrieved and stored locally in a database. This provides the capability to not only check current status information (e.g., flight status), but to also determine how the information has changed over time.

There are several ways in which plans can react to a monitoring event. First, a plan can use e-mail or fax operators to asynchronously notify interested parties about important updates to monitored data. Second, a plan can schedule or unschedule other agents in response to some condition. For example, once a flight has departed, the flight monitoring agent can schedule the *Flight-Connection* agent to run a few minutes before the scheduled arrival time.

Theseus agents are managed by a scheduling system that allows agents to be run once or at a fixed interval. The agent scheduler works by maintaining a database of the tasks to run and when they are scheduled to run next. Once scheduled, agents are launched at the appropriate time. Once they are run, the database is updated to reflect the next time the task is to be run. Since the Theseus plan language supports operators that can schedule and unschedule agents, this provides the ability to run new agents at appropriate times based on events in the world.

As an example information agent that monitors a data source, consider the plan for the *Flight-Status* agent shown in Figure 7. Initially, the agent executes a wrapper operator to retrieve the current flight status information. It then uses another online source to normalize the information based on the time zone of the recipient. The resulting normalized flight status information indicates that the flight is either arrived, cancelled, departed, or pending (waiting to depart). If the flight has been cancelled, the user is notified via the email operator. In this case, the flight needs no additional monitoring and the unschedule operator is used to remove it from the list of monitored flights. For departed flights the *Flight-Connection* agent is scheduled. For each pending flight, the agent must perform two actions. First, it checks if the arrival time of the flight is later than 5pm and if so it uses the fax operator to notify the hotel (it only does this once). Second, the agent compares the current departure time with the previously retrieved departure time. If they differ by a given threshold, the agent does three things: (a) it faxes a message to the car rental agency to notify them of the delay, (b) it updates its local database with the new departure time (to track future changes) and (c) it e-mails the user.

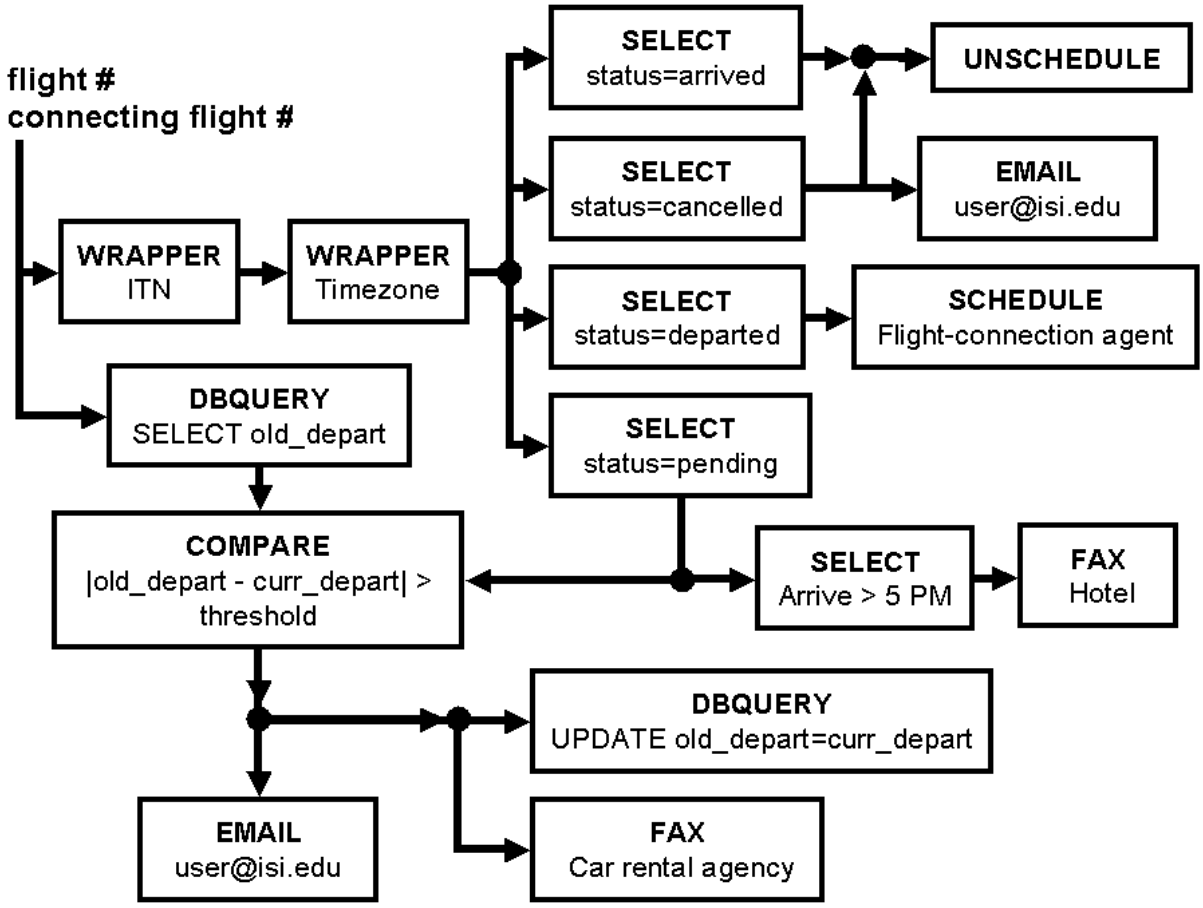


Figure 7: The Flight Status information monitoring agent

4.4 Efficiently Executing Agent Plans

Information agent plans are unique in two key respects. First, they tend to integrate data from multiple sources - for example, the *Flight-Status* agent might query multiple Internet real-time data sources to find out the in-flight status of a particular airplane. Second, they usually gather and monitor data from sources that are remote and deliver unpredictable performance - such as Internet web sites. Thus, information agent execution is often I/O-bound - with an agent spending the majority of its execution time waiting for replies from remote sources it has queried.

To efficiently execute plans that primarily integrate data from remote sources, Theseus employs a dataflow model of execution. Under this model, plan operators are arranged in a producer-consumer fashion, leading to partially ordered plans. Then, at run-time, operators can execute in parallel when their individual inputs have been received. This decentralized form of execution maximizes the amount of *horizontal parallelism*, or concurrency between independent data flows, available at run-time.

Theseus also supports the *streaming* of data between plan operators. Streaming enables consumer operators to receive data emitted by their producers as soon as possible. This, in

turn, enables these consumers to process this data immediately and communicate output to other consumers farther downstream as soon as possible. For example, consider an agent plan that fetches a large amount of data from two slow sources and joins this information together. Streaming enables the join to be executed as soon as possible on the available data, as it trickles in from either source. As a result, with streaming, producers and consumers may be processing the same logical set of data concurrently, resulting in a form of pipelined or *vertical parallelism* during execution.

5 Related Work

Most commercial systems for travel planning take the traditional approach of providing tools for selecting flights, hotel, and car rentals in separate steps. The only integrated approach is a system called MyTrip from XTRA On-line. Based on personal calendar information, the system automatically produces a complete plan that includes the flights, hotel and car rental. Once it has produced a plan, the user can then edit the individual selections made by the system. Unlike the *Travel Assistant*, the user cannot interactively modify the plan, such as constraining the airlines or departure airport. Also, MyTrip is limited to only the selection of flights, hotels, and car rentals. In addition to MyTrip, there exist some commercial systems (such as the one run by United Airlines) that provide basic flight status and notification. However, these systems do not actually track changes in the flight status over time (they merely notify passengers a fixed number of hours before flights) and they do not notify hotels about flight delays or suggest earlier flights or better connections when unexpected events (e.g., bad weather) occur.

In terms of constraint reasoning, there is a lot of research on constraint programming [Saraswat & van Hentenryck, 1995], but not much attention has been paid to the interplay between information gathering and constraint propagation and reasoning. Bressan and Goh [1997] have applied constraint reasoning technology to information integration. They use constraint logic programming to find relevant sources and construct an evaluation plan for a user query. In our system, the relevant sources have already been identified. In dynamic constraint satisfaction [Mittal & Falkenhainer, 1990], the variables and constraints present in the network are allowed to change with time. In our framework, this is related to interleaving the constraint satisfaction with the information gathering. Lamma et al. [1999] propose a framework for interactive constraint satisfaction problems (ICSP) in which the acquisition of values for each variable is interleaved with the enforcement of constraints. The information gathering stage in our constraint integration framework can also be seen as a form of ICSP. Their application domain is on visual object recognition, while our focus is on information integration

Our work on Theseus is related to two existing types of systems: *general agent executors* and *network query engines*. General agent executors, like RAPS [Firby, 1994] and PRS-Lite [Myers, 1996] propose highly concurrent execution models. The dataflow aspect of Theseus can be seen as another such model. The main difference is that execution in Theseus not only involves enabling operators but routing information between them as well. In this respect, Theseus shares much in common with several recently proposed network query engines [Ives *et al.*, 1999, Hellerstein *et al.*, 2000, Naughton *et al.*, 2001]. Like Theseus, these

systems focus in efficiently integrating web-based data. However, network query engines have primarily focused on performance issues; while Theseus also acts as an efficient executor, it is distinguished from these other query engines by (a) its novel operators that facilitate monitoring and asynchronous notification and (b) its modular dataflow agent language that allows a wider variety of plans to be built and executed.

6 Discussion

The travel planning and monitoring are fully functional systems that are in use today. The planner is not yet directly connected to a reservation system, but it is a very useful tool for helping to make the myriad of decisions required for planning a trip. Likewise, the monitoring agents are able to continually monitor all aspects of a trip and provide immediate notification of changes and cancellations. There are existing commercial systems that provide small pieces of these various capabilities, but the technology and application presented here is unique in providing a complete end-to-end solution that plans and then monitors all aspects of a trip. For example, the planning process will automatically produce the fax numbers required for use by the monitoring agents to notify the hotel and car rental agency. The current system is in use at the Information Sciences Institute and we plan to make this system more widely available on the Web as part of the Electric Elves Project [Chalupsky *et al.*, 2001].

One limitation of the current system is that the monitoring agents do not communicate problems or changes back to the travel planner. Ideally, if a flight is canceled one would want the system to re-book the traveler on another flight as soon as possible and then make any other needed changes to the itinerary based on the changes to the flight. Or if the price of a ticket declines to automatically rebook the ticket. The current system does not do this. We are working on the next generation of the travel planning component which will support this type of dynamic replanning of a trip based on changes in the world.

7 Acknowledgments

The research reported here was supported in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory under contract/agreement numbers F30602-01-C-0197, F30602-00-1-0504, F30602-98-2-0109, in part by the Air Force Office of Scientific Research under grant number F49620-01-1-0053, and in part by the Integrated Media Systems Center, a National Science Foundation Engineering Research Center, cooperative agreement number EEC-9529152. The U.S.Government is authorized to reproduce and distribute reports for Governmental purposes notwithstanding any copy right annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

References

[Barish & Knoblock, 2002] Barish, G., and Knoblock, C. A. 2002. Speculative plan execution for information gathering. In *Proceedings of the 6th International Conf. on AI Planning and Scheduling*.

[Barish *et al.*, 2000] Barish, G.; DiPasquo, D.; Knoblock, C. A.; and Minton, S. 2000. A dataflow approach to agent-based information management. In *Proceedings of the 2000 International Conference on Artificial Intelligence (IC-AI 2000)*.

[Bressan & Goh, 1997] Bressan, S., and Goh, C. H. 1997. Semantic integration of disparate information sources over the internet using constraint propagation. In *Workshop on Constraint Reasoning on the Internet (at CP97)*.

[Chalupsky *et al.*, 2001] Chalupsky, H.; Gil, Y.; Knoblock, C. A.; Lerman, K.; Oh, J.; Pynadath, D. V.; Russ, T. A.; and Tambe, M. 2001. Electric elves: Applying agent technology to support human organizations. In *Proceedings of the Thirteenth Innovative Applications of Artificial Intelligence Conference*.

[Firby, 1994] Firby, R. J. 1994. Task networks for controlling continuous processes. In *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems*.

[Hellerstein *et al.*, 2000] Hellerstein, J. M.; Franklin, M. J.; Chandrasekaran, S.; Deshpande, A.; Hildrum, K.; Madden, S.; Raman, V.; and Shah, M. A. 2000. Adaptive query processing: technology in evolution. *IEEE Data Engineering Bulletin* 23(2):7–18.

[Ives *et al.*, 1999] Ives, Z. G.; Florescu, D.; Friedman, M.; Levy, A.; and Weld, D. S. 1999. An adaptive query execution system for data integration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[Knoblock *et al.*, 2000] Knoblock, C. A.; Lerman, K.; Minton, S.; and Muslea, I. 2000. Accurately and reliably extracting data from the web: A machine learning approach. *IEEE Data Engineering Bulletin* 23(4).

[Knoblock *et al.*, 2001] Knoblock, C. A.; Minton, S.; Ambite, J. L.; Muslea, M.; Oh, J.; and Frank, M. 2001. Mixed-initiative, multi-source information assistants. In *Proceedings of the Tenth International World Wide Web Conference*.

[Lamma *et al.*, 1999] Lamma, E.; Mello, P.; Milano, M.; Cucchiara, R.; Gavanelli, M.; and Piccardi, M. 1999. Constraint propagation and value acquisition: Why we should do it interactively. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*.

[Mittal & Falkenhainer, 1990] Mittal, S., and Falkenhainer, B. 1990. Dynamic constraint satisfaction problems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 25–32.

[Myers, 1996] Myers, K. L. 1996. A procedural knowledge approach to task-level control. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*.

[Naughton *et al.*, 2001] Naughton, J. F.; DeWitt, D. J.; Maier., D.; et al. 2001. The niagara internet query system. *IEEE Data Engineering Bulletin* 24(2):27–33.

[Saraswat & van Hentenryck, 1995] Saraswat, V., and van Hentenryck, P., eds. 1995. *Principles and Practice of Constraint Programming*. Cambridge, MA: MIT Press.

Appendix C:

THE ARIADNE APPROACH TO WEB-BASED INFORMATION INTEGRATION[1]

CRAIG A. KNOBLOCK, STEVEN MINTON, JOSE LUIS AMBITE,
NAVEEN ASHISH, ION MUSLEA, ANDREW G. PHILPOT, and SHEILA TEJADA

Information Sciences Institute, Integrated Media Systems Center,
and Department of Computer Science
University of Southern California
4676 Admiralty Way,
Marina del Rey, CA 90292

Abstract

The Web is based on a browsing paradigm that makes it difficult to retrieve and integrate data from multiple sites. Today, the only way to do this is to build specialized applications, which are time-consuming to develop and difficult to maintain. We have addressed this problem by creating the technology and tools for rapidly constructing information agents that extract, query, and integrate data from web sources. Our approach is based on a uniform representation that makes it simple and efficient to integrate multiple sources. Instead of building specialized algorithms for handling web sources, we have developed methods for mapping web sources into this uniform representation. This approach builds on work from knowledge representation, databases, machine learning and automated planning. The resulting system, called Ariadne, makes it fast and easy to build new information agents that access existing web sources. Ariadne also makes it easy to maintain these agents and incorporate new sources as they become available.

1 Introduction

The amount of data accessible via the Web and intranets is staggeringly large and growing rapidly. However, the Web's browsing paradigm does not support many information management tasks. For instance, the only way to integrate data from multiple sites is to build specialized applications by hand. These applications are time-consuming and costly to build, and difficult to maintain.

This paper describes Ariadne,[2] a system for extracting and integrating data from semi-structured web sources. Ariadne enables users to rapidly create *information agents* for the Web. Using Ariadne's modeling tools, an application developer starts

[1]This article appeared in the International Journal of Cooperative Information Systems, Special Issue on Intelligent Information Agents: Theory and Applications, 10(1/2), pg 145-169, 2001. This article is an extended version of the article originally published in AAAI'98 [20]

[2]In Greek mythology, Ariadne gave Theseus the thread that let him find his way out of the Minotaur's labyrinth.

with a set of web sources – semi-structured HTML pages, which may be located at multiple web sites – and creates a unified view of these sources. Once the modeling process is complete, an end user (who might be the application developer himself) can issue database-like queries as if the information were stored in a single large database. Ariadne's query planner decomposes these queries into a series of simpler queries, each of which can be answered using a single HTML page, and then combines the responses to create an answer to the original query.

The modeling process enables users to integrate information from multiple web sites by providing a clean, well-understood representational foundation. Treating each web page as a relational information source – as if each web page was a little database – gives us a simple, uniform representation that facilitates the data integration. The representation is quite restricted, but we compensate for that by developing intelligent modeling tools that help application developers map complex web sources into this representation.

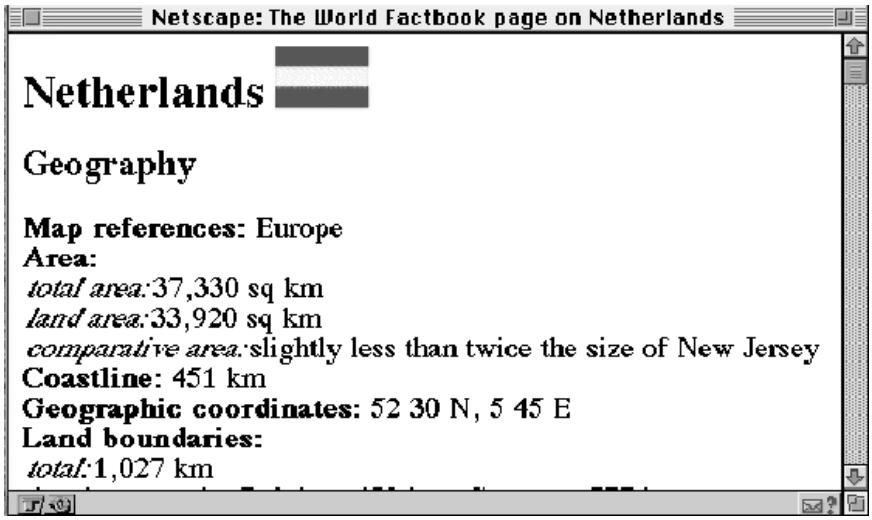


Figure 1: A CIA Factbook Page

We will illustrate Ariadne by considering an example application that involves answering queries about the world's countries. An excellent source of data is the CIA World Factbook, which has an HTML page for each country describing that country's geography, economy, government, etc. The top of the Factbook page for the Netherlands is shown in Figure 1.[3] Some of the many other relevant sites include the NATO site, which lists the NATO member countries (shown in Figure 2), and the World Governments site, which lists the head of state and other government officers for each country (shown in Figure 3). Consider queries such as "What NATO countries have populations less than 10 million?" and "List the heads of state of all the countries in the Middle East". Since these queries span multiple countries and require combining information from multiple sources, answering them by hand is time consuming. Ariadne allows us to rapidly put together a new application that can answer a wide range of queries by extracting and integrating data from prespecified web sources.

[3]All the web sources in our examples are based on real sources that Ariadne handles, but we have simplified some of them here for expository purposes.

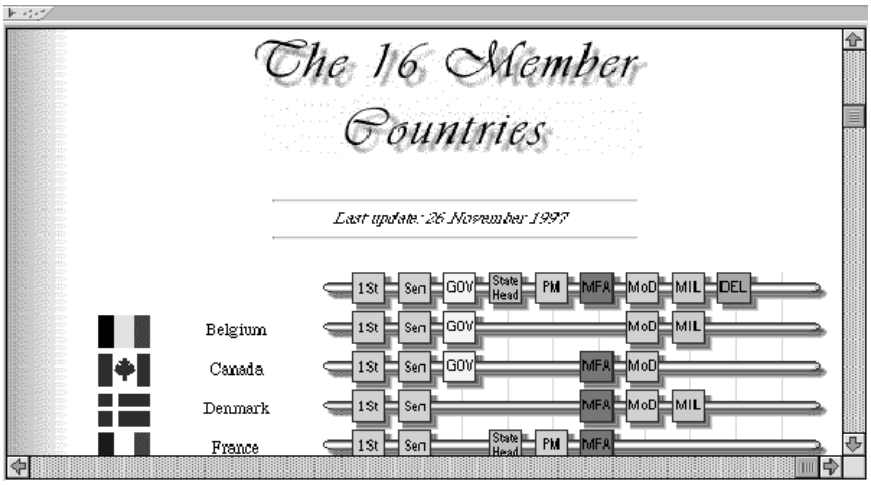


Figure 2: NATO Members Page

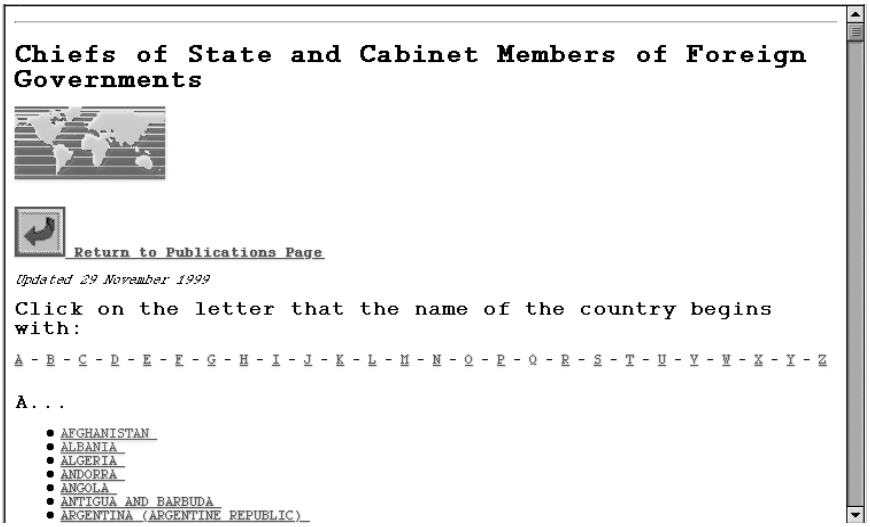


Figure 3: World Governments Page

In the following section we describe our basic approach to query planning, where a unifying domain model is used to tie together multiple information sources. We then describe the details of our modeling approach: how we represent and query individual web pages, how we represent the relationships among multiple pages in a single site, how we integrate data that spans multiple sites, and how we represent and materialize data locally to optimize an application. In each section, we also describe the methods that are used in modeling and query processing, and how the uniform representational scheme supports these methods.

2 Approach to Information Integration

The Ariadne integration framework consists of the following components:

- A model of the application domain,

- A description of the information sources in terms of this model,
- Wrappers that provide uniform access to the information sources so that they can be queried as if they were relational databases, and
- A query planner that dynamically determines how to efficiently process a user query given the set of available information sources.

As we describe in this paper, these components provide the infrastructure to build a complete web-based information integration system. For example, navigating through a web site is simply a matter of creating wrappers for the navigation pages, representing these pages, and letting the query planner generate the appropriate navigation plans. Similarly, resolving naming inconsistencies across sites is addressed by building a new information source that provides the mapping, modeling this information source, and using the query planning to generate plans that use these mappings when needed. Finally, locally storing data to optimize plans can be done simply by creating a new information source with the cached data, modeling this source, and relying on the query planning to use this cached information when needed.

In this section we provide an overview of Ariadne's integration model and how it facilitates efficient query planning. In later sections we will show how this uniform model supports the requirements of a complete information integration system for the web.

Ariadne's approach to information integration is an extension of the SIMS mediator architecture [5, 6, 19]. SIMS was designed for structured information sources such as databases and knowledge bases (and to some extent output from programs). In Ariadne, we extend the SIMS approach to semi-structured sources such as web sources by using wrappers. Also, database applications typically involve only a small number of databases, while web applications can involve accessing many more sources. Since the SIMS planner did not scale well to large numbers of sources, for Ariadne we developed an approach capable of efficiently constructing large query plans by precompiling part of the integration model and using a local search method for query planning [4].

2.1 Integration Model

In SIMS and Ariadne the mediator designer defines a *domain model*, which is an ontology of the application domain that integrates the information in the sources and provides a single terminology over which the user poses queries. The domain model is represented using the Loom knowledge representation system [24]. Each information source is defined to be equivalent to a class description in the domain model. Thus, Ariadne uses a form of local-as-view source descriptions, cf. [28]. This approach facilitates the addition of new sources to the mediator, since the new source definitions do not interact with the previous ones.

As an example of a domain model consider Figure 4. For simplicity of exposition, this model assumes that the information in the three web sites described earlier, the CIA World Factbook, the World Governments site, and the NATO members page, is available in three separate databases, along with a fourth database containing a map for each country (later we show the modeling when the information is available from web sources, see Figure 11). The model contains four classes with some relations between them. For example, 'NATO Country' is a subclass of 'Country', and 'Country' has a relation called 'Head-of-State' which points to a class with the same name. Each domain class has a set of attributes. For example, some of the attributes of the

'Country' class are total area, latitude, population, etc. We use the domain model to describe the contents of each information source. For example, the figure shows that the CIA Factbook is a source for information about Countries, and the World Governments database is a source for Heads of State. A source may provide only a subset of the attributes for a given domain class, so the system may need to combine several sources to obtain all the desired attributes. For example, if the user requests the total area and a map of a country, the system must retrieve the total area from the CIA World Fact Book and the map from the Map database.

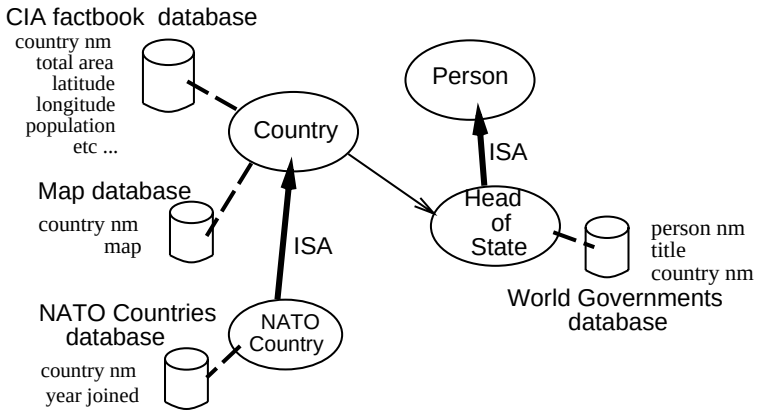


Figure 4: Domain Model with Database Sources

2.2 Query Processing

Queries are presented to the system in terms of the domain model. For example, a query might be "List the heads of state of all the countries whose population is less than ten million."[4] The system then decomposes the query into subqueries on the individual sources, such as the World Governments and Factbook sources, producing a query plan consisting of relational operators (i.e., joins, selects, projects, etc.) and access operations to the sources.

To improve the efficiency of query planning, we used two techniques. First, the source descriptions are compiled off-line into a more convenient form. Second, we implemented a transformational query planner that explores the space of query evaluation plans using local search.

2.2.1 Integration Axiom Precompilation

To facilitate query planning, we developed an algorithm [5] to compile the local-as-view source descriptions into global-as-view integration axioms. Each integration axiom specifies an alternative combination of sources that produces a maximal set of attributes for a domain class. Once the integration axioms are compiled, finding the

[4]We use English translations of the queries for clarity. In the system the queries can be expressed using either SQL or the Loom query language.

sources relevant for a user domain query is straightforward. For each domain class in the query the system looks up the integration axioms for that class and its attributes mentioned in the query. The body of each integration axiom is a formula containing only source terms that can be substituted by the corresponding domain class in the query. Compiling axioms *a priori* is a space-for-time optimization, allowing the system to amortize the cost of producing them over all queries, thus avoiding repetition of costly run-time search. This section presents the highlights of the approach, see [5] for details.

Ariadne generates the integration axioms from the source descriptions and the structure of the domain model. The axiom compilation algorithm is based on the iterative application of a set of inference rules. The rules are applied in parallel, constructing a generation of novel integration axioms from existing axioms. The process is repeated until quiescence. The five rules are:

- *Direct Rule*: Inverts the source descriptions.
- *Covering Rule*: Exploits the covering relationships in the domain model (when a class is defined as the union of its subclasses, the subclasses constitute a covering of the class).
- *Definition Rule*: Exploits the constraints in the definition of a domain class.
- *Inherit Rule*: Exploits the inheritance of superclass attributes via shared keys.
- *Compose Rule*: Combines axioms on a given class to provide additional attributes.

The compiled axioms from the domain of Figure 4 are shown in Figure 5. In the first generation the source descriptions are inverted and installed as axioms via the *direct* rule (the axioms *1.1*, *1.2*, *1.3*, and *1.4*). Then, the sources are composed iteratively to produce more attributes for each domain class. In the second generation, the *inherit rule* produces axioms *2.1* and *2.2* by adding the NATO-db source to the body of axioms *1.1* and *1.2* which specilizes them for NATO countries and provides the additional 'map' attribute. Also, the *compose* rule generates axiom *2.3* by joining the CIA Factbook database and the Map database over the common key attribute 'country-nm'. Finally, in the third generation, the inherit rule constructs axiom *3.1* by adding the NATO-db source to axiom *2.3* in order to produce the maximal number of attributes that the available sources provide for the 'NATO Country' class. In this domain, rule application takes three generations until quiescence producing the 8 axioms of Figure 5.

2.2.2 Query Planning

Once the integration axioms have been compiled, Ariadne is ready to accept user queries expressed over terms of the domain model. Query planning for each user query follows two steps. First, the query is parsed, simplified, and rewritten so that each class mentioned in the query is the most specific according to the definitions in the domain model. This ensures that the appropriate integration axioms are used during query planning. For example, consider an extension to Figure 4 which includes the class 'Military Leader', defined as the subclass of 'Head of State' where the title is equal to "general", "colonel", etc. An input query that asks for information about a 'Head of State' whose title is "general" will find all the relevant axioms associated with the class 'Military Leader' as opposed to the class 'Head of State' that is mentioned in the query.

Country(cn ta lat long pop)	⇔	CIA-db(cn ta lat long pop)	*1.1*
Country(cn map)	⇔	Map-db(cn map)	*1.2*
Country(cn ta lat long pop map)	⇔	CIA-db(cn ta lat long pop) ∧ Map-db(cn map)	*2.3*
NATO-Country(cn year)	⇔	NATO-db(cn year)	*1.3*
NATO-Country(cn ta lat long pop year)	⇔	CIA-db(cn ta lat long pop) ∧ NATO-db(cn year)	*2.1*
NATO-Country(cn map year)	⇔	Map-db(cn map) ∧ NATO-db(cn year)	*2.2*
NATO-Country(cn ta lat long pop map year)	⇔	CIA-db(cn ta lat long pop) ∧ Map-db(cn map) ∧ NATO-db(cn year)	*3.1*
Head-of-State(pn cn title)	⇔	WorldGov-db(pn cn title)	*1.4*

Figure 5: Compiled Integration Axioms

Second, the query is optimized using a transformational query planner [4, 3] based on the Planning by Rewriting paradigm [2, 1]. The query planner first constructs an initial query evaluation plan based on a depth-first parse of the query. This initial plan is possibly suboptimal, but it is generated very efficiently. Then, the plan is iteratively transformed using a set of rewriting rules in order to optimize the plan cost. The rewriting rules are derived from properties of the relational algebra, the distributed environment, and the integration axioms in the application domain. The space of plan rewritings is explored efficiently using local search methods. During the rewriting process, the planner considers the different sources, operators, and orders of the operators that can be used to answer the query.

The query planner has a modular, declarative, and extensible architecture. The initial plan generator, the cost metric, the set of rewriting rules, and the strategy used during the rewriting search, are all modules that can be extended or replaced independently. Since our query planner is based on a domain-independent approach to planning, it is extensible in a principled way and very flexible. The specification of both the plan operators and the plan rewriting rules is declarative.

As an example, consider the processing required to retrieve the names of all NATO countries and the year they joined NATO for those countries that have a population of less than 10 million. In this case, the relevant axiom that provides the population of NATO countries and the year of incorporation (using a minimal set of sources) is axiom *2.1* in Figure 5 (if there were several alternative axioms for that information, the planner would consider them during the search). Based on this axiom, assume that the planner constructs the initial plan of Figure 6. This plan is suboptimal since it retrieves the names and population for all countries from the Factbook source, the names and year from the NATO source, and then joins both relations locally, which is very costly since the Factbook source is quite large. Moreover, the selection on population is done after the join, instead of being used to reduce the number of tuples that participate in the join. The query planner rewrites this initial plan producing the optimized plan of Figure 7. The optimized plan first retrieves the name and year of the NATO countries, projects the country names, and passes NATO country names so

that only the population of NATO countries is retrieved from the CIA World Factbook source. In addition the selection on population has been placed immediately after the retrieval from the Factbook source to further reduce the number of tuples participating in the join.

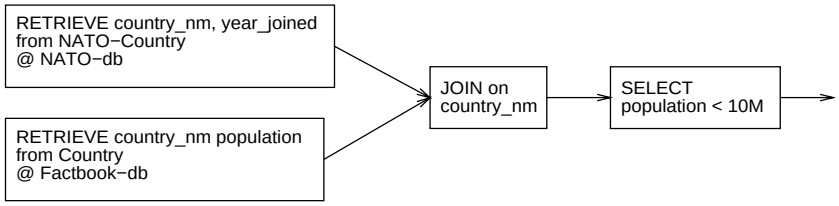


Figure 6: A Suboptimal Initial Query Plan

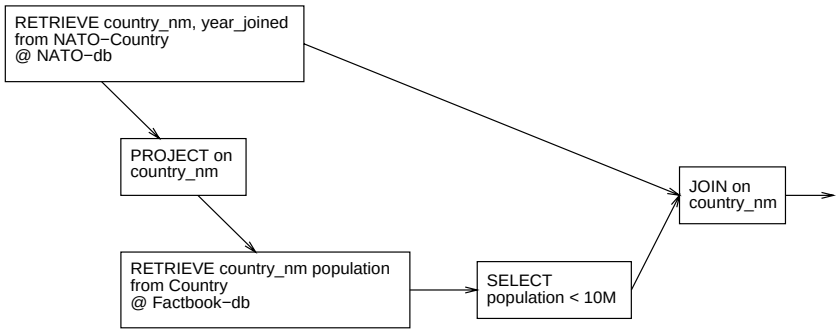


Figure 7: An Optimized Query Evaluation Plan

Ariadne's uniform integration model is expressive enough to encompass a wide variety of web sources but simple enough to allow for efficent query planning. In the following sections, we discuss how, based on this simple model, we provide a coherent set of solutions for each level of the problem of integrating information on the web. First, we discuss how we model and automatically learn wrappers for individual web pages. Second, we describe the modeling that allows our query planner to generate plans that navigate among pages. Third, we present techniques to identify entities across sites so that they can be integrated. Fourth, we describe further performance optimization by selectively materializing data. Finally, we describe some of the applications that have been developed using Ariadne and assess the approach.

3 Modeling the Information on a Page

The previous section described how the planner decomposes a complex query into simple queries on individual information sources. To treat a web page as an information source so that it can be queried, Ariadne needs a wrapper that can extract and return the requested information from that type of page. While we cannot currently create such wrappers for unrestricted natural language texts, many information sources on the Web are *semistructured*. A web page is semistructured if information on the page can be located using a concise formal grammar, such as a context-free grammar. Given such a grammar, the information can be extracted from the source without recourse to sophisticated natural language understanding techniques. For example, a wrapper

for pages in the CIA Factbook would be able to extract fields such as the Total Area, Population, etc. based on a simple grammar describing the structure of Factbook pages.

Our goal is to enable application developers to easily create their own wrappers for web-based information sources. To construct a wrapper, we need both a semantic model of the source that describes the fields available on that type of page and a syntactic model, or grammar, that describes the page format, so the fields can be extracted. Requiring developers to describe the syntactic structure of a web page by writing a grammar by hand is too demanding, since we want to make it easy for relatively unsophisticated users to develop applications. Instead, Ariadne has a "demonstration-oriented user interface" (DoUI) where users show the system what information to extract from example pages. Underlying the interface is a machine learning system for inducing grammar rules.

Figure 8 shows how an application developer uses the interface to teach the system about CIA Factbook pages, producing both a semantic model and a syntactic model of the source. The screen is divided into two parts. The upper half shows an example document, in this case the Netherlands page. The lower half shows a semantic model, which the user is in the midst of constructing for this page. The semantic model in the figure indicates that the class Country has attributes such as Total Area, Coastline, Latitude, Longitude, etc. The user constructs the semantic model incrementally, by typing in each attribute name and then filling in the appropriate value by cutting and pasting the information from the document. In doing so, the user actually accomplishes two functions. First, he provides a name for each attribute. Notice that he can choose the same names as used in the document (e.g., "Total area") or he can choose new/different names (e.g., "Latitude"). As we will explain later, the attribute names have significance, since they are the basis for integrating data across sources.

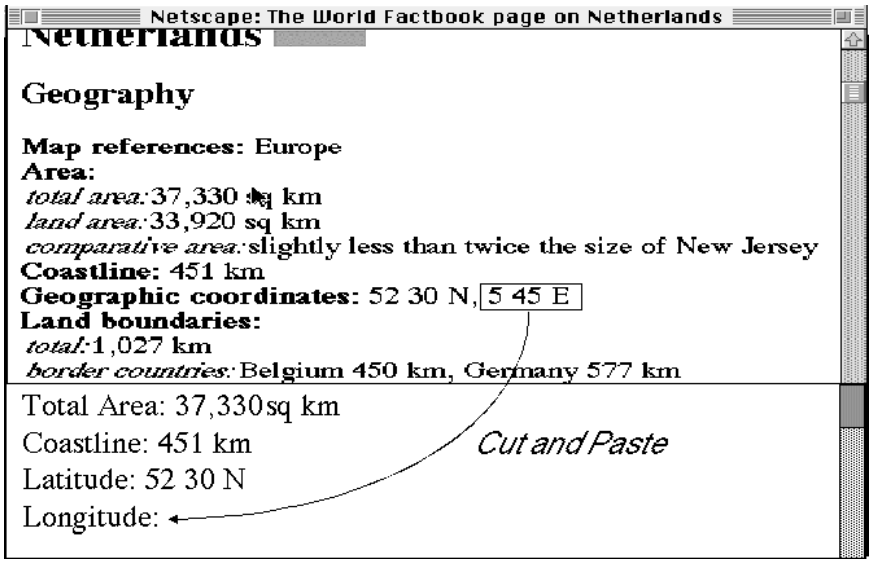


Figure 8: Creating a Wrapper by Demonstration

The second function achieved by the user's demonstration is to provide examples so that the system can induce the syntactic structure of the page. Ideally, after the user has picked out a few examples for each field, the system will induce a grammar sufficient for extracting the required information for all pages of this type. Unfortunately,

grammar induction methods may require many examples, depending on the class of grammars being learned. However, we have observed that web pages have common characteristics that we can take advantage of, so that a class of grammars sufficient for extraction purposes can be rapidly learned in practice.

More specifically, we can describe most semistructured web pages as *embedded catalogs*. A *catalog* is either a homogeneous list, such as a list of numbers, (1,3,5,7,8), or a heterogeneous tuple, such as a 3-tuple consisting of a number, a letter, and a string, (1,A,"test"). An *embedded catalog* (or, for short, $\mathcal{EC}$) is a catalog where the items themselves can be catalogs. As an example, consider the fragment of the CIA Factbook page shown in Figure 8. At the top level, it can be seen as a 9-tuple that consists of Map References, Total Area, Land Area, Comparative Area, Coastline, Latitude, Longitude, Total Land Boundaries, and Border Countries. Furthermore, the Border Countries represent an embedded list of 2-tuples that contain a Country Name and a Border Length. Note that the illustrative page description above contains *all* the fields in the document, while in practice, the user can identify only the items of interest for a particular application. For instance, a user can model the CIA Factbook page as a 4-tuple that consists of the Total Area, Latitude, Longitude, and Border Countries, where the last field is an embedded list of 2-tuples that contain a Country Name and a Border Length.

An embedded catalog is a structured description of the information on a page, and can be (trivially) converted into an XML view of the document, as illustrated in Figure 9.

```
<!DOCTYPE document [
 <!ELEMENT document ( TotalArea, Latitude, Longitude,
                      Neighbors_LIST )>
 <!ELEMENT TotalArea (#PCDATA)>
 <!ELEMENT Latitude (#PCDATA)>
 <!ELEMENT Longitude (#PCDATA)>
 <!ELEMENT Neighbors_LIST ( Neighbor* )>
 <!ELEMENT Neighbor (Name, BorderLength)>
 <!ELEMENT Name (#PCDATA)>
 <!ELEMENT BorderLength (#PCDATA)> ] >
```

Figure 9: **Sample XML description of an embedded catalog.**

Besides being used as *data schema* in the integration process, the $\mathcal{EC}$ description of a document plays another important role: in Ariadne, we use a document's $\mathcal{EC}$ to extract the data in a hierarchical manner. For instance, in order to extract all the neighboring countries from a document, we begin by extracting the Border Countries from the whole document; then we iterate through this list, and break it down to individual 2-tuples; finally, from such tuples, we extract each individual Country Name. In other words, if we look at $\mathcal{EC}$ as a tree-like structure describing the embedded data, in order to extract a relevant item we must successively extract each of its ancestors from their respective parents in the tree. Our approach has a major advantages: it transforms a potentially hard problem (i.e., extracting *all* items from an arbitrarily complex document) into a set of simpler ones (i.e., extracting *one* individual item from its parent in the $\mathcal{EC}$). This is particularly appealing when a document contains a large number of items and

multiple levels of embedded data (e.g., list within lists).

Because web pages are intended to be human readable, special markers often play a role identifying the beginning or ending of an item in an embedded catalog, separating items in a homogeneous list, and so on. These distinguishing markers can be used as landmarks for locating information on a page. A *landmark grammar* describes the position of a field via a sequence of landmarks, where each landmark is a sequence of tokens and wildcards (e.g., $Number$, $CapitalizedWord$, $AllCaps$, etc.). For example, to find the beginning of the longitude, we can use the rule

$$\mathbf{R1} = SkipTo(\texttt{Geographic coordinates})\ SkipTo(\texttt{,})$$

which has the following meaning: start from the beginning of the document and skip everything until you find the landmark `Geographic coordinates`; then, again, ignore all tokens until you encounter the first comma. Similarly, we can use the rule

$$\mathbf{R2} = SkipTo(\ \texttt{
 Land boundaries})$$

to identify the end of the longitude field (`<br>` is the HTML tag that forces the line break, and, consequently, it is not displayed by the browser in Figure 8).

In order to fully define a wrapper, one needs to provide the embedded catalog, together with one extraction rule for each field and one additional iteration rule for each list in the $\mathcal{EC}$ (iteration rules are applied *repeatedly* to the content of the list in order to extract all individual tuples). Our recent work [26] shows that in practice, a subclass of landmark grammars (i.e., linear landmark grammars) can be learned rapidly for a variety of web pages using a greedy covering algorithm. There are several reasons for this. First, because web pages are intended to be human readable, there is often a *single* landmark that distinguishes or separates each field from its neighbors. Therefore, the length of the grammar rules to be learned will usually be very small, and learning will be easy in practice. Second, during the demonstration process, users traverse a page from top-to-bottom, picking out the positive examples of each field. Any position on the page that is not marked as a positive example is implicitly a negative example. Thus, for every positive example identified by the user, we obtain a huge number of negative examples that the covering algorithm can use to focus its search.

The empirical evaluation of STALKER [26], our wrapper induction system, shows that in most of the cases our system learns perfect extraction rules (i.e., 100% accuracy) based on just a handful of examples. We tested STALKER on 30 information sources, which required the induction of 206 different rules. In 182 cases STALKER generated a perfect rule (in most of the cases based on just a couple of labeled examples), and 18 other rules had an accuracy above 90% based on as few as 10 training examples. On the same 30 information sources, WIEN [21], which was the first wrapper induction system, requires one to two orders of magnitude more labeled examples in order obtain a similar or worse performance.

There are several differences between the approaches taken by STALKER and WIEN. First of all, WIEN's approach to handling documents with multiple levels of embedded data turned out to be impractical: even for the domains in which such a wrapper exists, the learning algorithm failed to find it. Second, WIEN uses a very simple extraction language. By assuming that all the fields are *always* present and in exactly the same order, WIEN is capable of learning the rules extremely fast, provided that they exist. On the other hand, these assumptions make it impossible to wrap more complicated

sources. Last but not least, the same simplicity of the extraction language, together with the fact that WIEN does not extract the sibling fields independently of each other, leads to failure to wrap sources from which STALKER finds perfect rules for most of the items, and slightly imperfect ones for the remaining fields.

A quite different approach to wrapper induction is the one used in SoftMealy [16]. This system induces extraction rules expressed as finite transducers, and it addresses most of the WIEN's shortcomings. However, its empirical evaluation is quite sketchy, which makes it hard to compare with WIEN and STALKER. There are three other recent systems that are focusing on learning extraction rules from online documents: SRV [13], RAPIER [11], and WHISK [27]. Even though these approaches are mostly concerned with extracting data from natural language text, they could be also applied to some simple wrapper induction problems.

The modeling tool we have described enables unsophisticated users to turn web pages into relational information sources. But it has a second advantage as well. If the format of a web source changes in minor respects, the system could induce a new grammar by reusing examples from the original learning episode, without any human intervention (assuming the underlying content has not changed significantly). This is a capability we are currently exploring.

4 Modeling the Information in a Site: Connections between Pages

The previous section showed how Ariadne extracts information from a web page to answer a query. However, before extracting information from a page, Ariadne must first locate the page in question. Our approach, described in this section, is to model the information required to "navigate" through a web site, so that the planner can automatically determine how to locate a page.

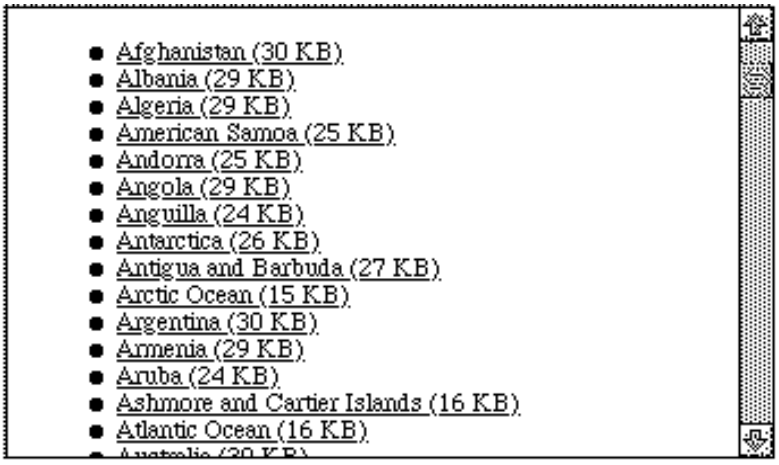


Figure 10: CIA Factbook Index

For example, consider a query to our example information agent asking for the population of the Netherlands. To extract the population from the Factbook's page on the Netherlands, the system must first find the URL for that page. A person faced with the same task would look at the index page for the Factbook, shown in Figure 10,

which lists each country by name together with a hypertext link to the page in question. In our approach, Ariadne does essentially the same thing. The index page serves as an information source that provides a URL for each country page. These pages in turn serve as a source for country-specific information.

To create a wrapper for the index page, the developer uses the approach described in the last section, where we illustrated how a wrapper for the Factbook's country pages is created. There is only one difference: this wrapper only wraps a single page, the index page. The developer creates a semantic model indicating that the index page contains a list of countries, each with two attributes, country-nm and country-URL.[5] The learning system induces a grammar for the entire page after the developer shows how the first few lines in the file should be parsed.

As the wrappers for each source are developed, they are integrated into the unifying domain model. Figure 11 shows the domain model for the completed geopolitical agent. (Notice that we have substituted web source wrappers for the hypothetical databases used previously.) To create the domain model, the developer specifies the relationship between the wrappers and the domain concepts. For instance, the developer specifies that the Factbook country wrapper and the Factbook index wrapper are both information sources for "country" information, and he identifies which attributes are keys (i.e., unique identifiers). In the example, "country-nm" and "country-URL" are both keys. Binding constraints specify the input and output of each wrapper (shown by the small directional arrows in Figure 11). The country page wrapper takes a country-URL, and acts as a source for "total area", "population", "latitude", etc. The index wrapper takes a country name[6] and acts as a source for "country-URL". Given the domain model and the binding constraints, the system can now construct query plans. For instance, to obtain the population of a country given its name, the planner determines that the system must first use the country name to retrieve the country-URL from the index page wrapper, and then use the country-URL to retrieve the population data from the country page wrapper.

Explicitly modeling 'navigation' pages, such as the Factbook index, as information sources enables us to reuse the same modeling tools and planning methodology underlying the rest of the system. The approach works well in part because there are only two common types of navigation strategies used on the Web – direct indexing and form-based retrieval. We have already seen how index pages are handled; form-based navigation is also straightforward. A wrapper for an HTML form simply mimics the action of the form, taking as input a set of attributes, each associated with a form parameter name, and communicating with the server specified in the form's HTML source.

When the resulting page is returned, the wrapper extracts the relevant attributes in the resulting page. Imagine, for instance, a form-based front end to the Factbook, where the user types in a country name and the form returns the requested country page. To create a wrapper for this front end, the developer would first specify that the parameter associated with the type-in box would be filled by a "country-nm". He would then specify how the system should extract information from the page returned by the form using the approach described in the last section.

The Factbook example described in this section illustrates our basic approach to

[5]During the demonstration, a check box is used to extract a URL from a hyperlink, as opposed to grabbing text.

[6]No URL is needed as input to the index page wrapper since the URL of the index page is a constant.

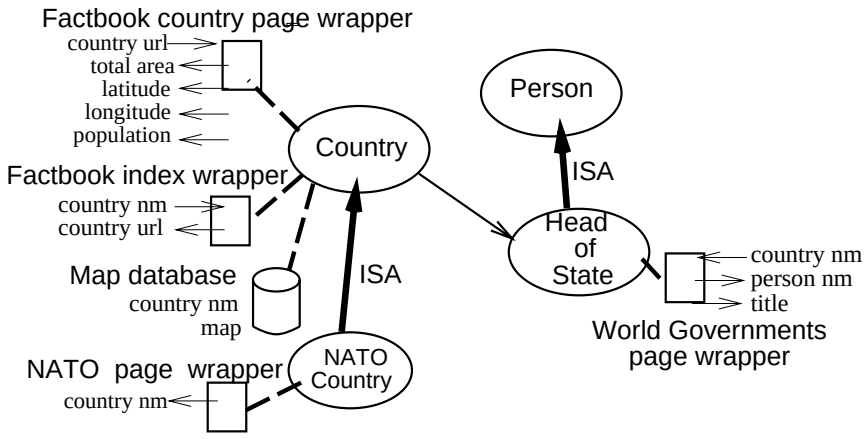


Figure 11: Domain Model with Web Sources

modeling navigation pages. Many web sites are more complex than the Factbook. The approach still works, but the models become more involved. For instance, indexes can be hierarchical, in which case each level of the hierarchy must be modeled as an information source. Imagine the top-level Factbook index was a list of letters, so that clicking on a letter "C" would produce an index page for countries starting with "C" (a "subindex"). We would model this top level index as a relation between letters and subindex-URL's. To traverse this index, we also need an information source that takes a country name and returns the first letter of the name (e.g., a string manipulation program). Thus, altogether four wrappers would be involved in the navigation process, as shown in Figure 12. Given a query asking for the Netherlands' population, the first wrapper would take the name "Netherlands", call the string manipulation program, and return the first letter of the name, "N". The second wrapper would take the letter "N", access the top level index page, and return the subindex-URL. The third wrapper would take the subindex-URL and the country name, access the subindex page for countries starting with "N", and return the country-URL. Finally, the last wrapper would take the country-URL and access the Netherlands page. The advantage of our approach is that all these wrappers are treated uniformly as information sources, so the query planner can automatically determine how to compose the query plan. Furthermore, the wrappers can be semi-automatically created via the learning approach described earlier, except for the string manipulation wrapper, which is a common utility.

5 Modeling Information Across Sites

Within a single site, entities (e.g., people, places, countries, companies, etc.) are usually named in a consistent fashion. However, across sites, the same entities may be referred to with different names. For example, the CIA Factbook refers to "Burma" while the World Governments site refers to "Myanmar". Sometimes formatting conventions are responsible for differences, such as "Denmark" vs. "Denmark, Kingdom of". To make sense of data that spans multiple sites, we need to be able to recognize

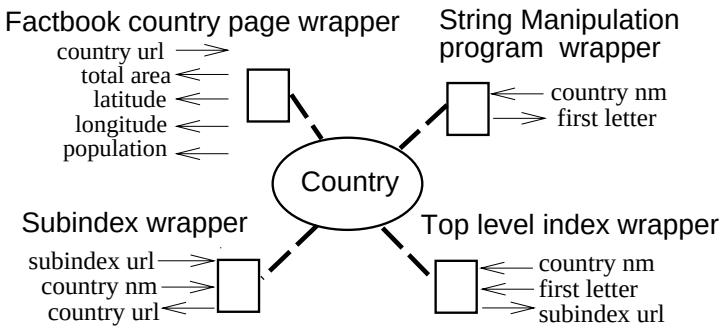


Figure 12: Domain Model with Hierarchical Index

and resolve these differences.

Our approach is to select a primary source for an entity's name and then provide a mapping from that source to each of the other sources where a different naming scheme is used. An advantage of the Ariadne architecture is that the mapping itself can be represented as simply another wrapped information source. One way to do this is to create a *mapping table*, which specifies for each entry in one data source what the equivalent entity is called in another data source. Alternatively, if the mapping is computable, it can be represented by a *mapping function*, which is a program that converts one form into another form.

Figure 13 illustrates the role of mapping tables in our geopolitical information agent. The Factbook is the primary source for a country's name. A mapping table maps each Factbook country name into the name used in the World Governments source (i.e., WG-country-nm). The mapping source contains only two attributes, the (Factbook) country name and the WG-country-nm. The NATO source is treated similarly. So, for example, if someone wanted to find the Heads of State of the NATO countries, the query planner would retrieve the NATO country names from the NATO wrapper, map them into (Factbook) country names using the NATO mapping table, then into the World Government country names using the World Governments mapping table, and finally retrieve the appropriate heads of state from the World Governments wrapper.

We have developed a semi-automated method for building mapping tables and functions by analyzing the underlying data in advance. The method attempts to pair each entity in one source with a corresponding entity (or entities, in some cases) in another source. The basic idea is to use information retrieval techniques to provide an initial mapping (following [12]), and then to apply machine learning techniques to improve the mapping. The initial mapping matches entities from two sources based on their textual similarity. For example, "Denmark" and "Denmark, Kingdom of" are assumed to refer to the same entity because both names include "Denmark", an infrequently-occurring word. In the subsequent learning phase, the system learns two types of rules to help improve/verify the initial mapping. *Transformation rules* identify textual transformations like acronyms, abbreviations, and phrase orderings that are common in the domain. For instance, the system can learn that "Rep" is a commonly used abbreviation for "Republic", or that one source commonly employs acronyms, or that one source represents person names as "LastName, Firstname", while the other uses "FirstName LastName". The system also learns *Mapping rules* which are used

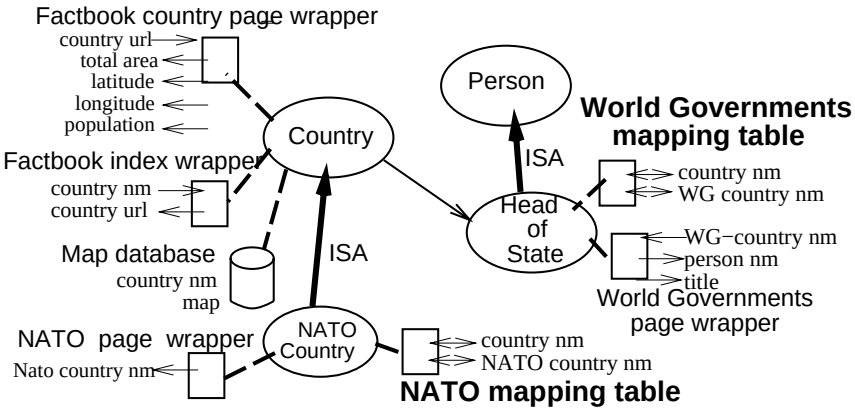


Figure 13: Domain Model with Mapping Tables

when we can compare entities along multiple attributes.

For example, consider a multi-year Factbook application which includes yearly versions of the Factbook (each year a new version of the CIA Factbook is released), as shown in Figure 14. Sometimes countries in the new Factbook have new names, or countries merge or split. These name confusions can often be resolved by using the attributes containing geographical information, e.g., land area, latitude and longitude. These attributes stay constant over time and are unique to a country, so they are good indicators of a mapping. Mapping rules specify the importance of each attribute when computing a mapping.

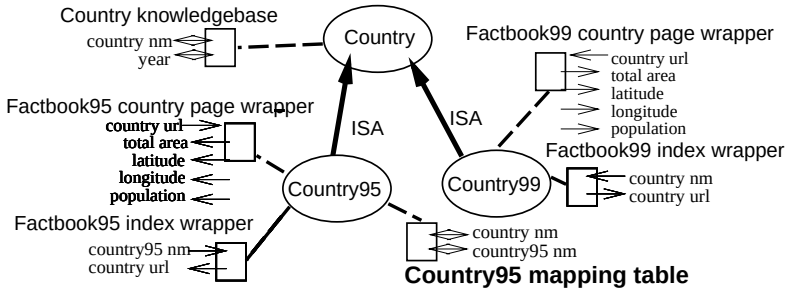


Figure 14: Multi-Year Factbook Model

We are prototyping an active learning method for learning transformation rules and mapping rules. In our approach, a human is asked to verify some of the pairs in the initial mapping, i.e., to indicate whether the pairs are correctly or incorrectly matched. Then the system attempts to learn new rules, and selects additional pairs for the human to verify. The system selects the pairs that will be most valuable for confirming/disconfirming the hypotheses explored by the learning algorithm. The goal is to obtain a mapping for which the system is highly confident, while minimizing the time the human must spend.

6 Performance Optimization by Selective Materialization

An issue with building applications using Ariadne is that the speed of the resulting application is heavily dependent on the individual web sources. The response time may be high even when the query planner generates high-quality information integration plans. This is because for some queries a large number of pages may have to be fetched from remote Web sources and some sources may be very slow. In this section we describe our approach to optimizing agent performance by locally materializing selected portions of the data. The materialized data is simply a locally stored relation that is handled using the same uniform representation and query planning techniques used throughout the system.

The brute force approach would be to simply materialize all the data in all the Web sources being integrated. However this is impractical for several reasons. First the sheer amount of space needed to store all this data locally could be very large. Second, the data might get updated at the original sources and the maintenance cost for keeping all the materialized data consistent could be very high. Our approach is thus to materialize data *selectively*. In this approach to optimizing performance by selectively materializing data there are two key issues that must be addressed.

1. What is the overall framework for materializing data, i.e., how do we represent and use the materialized data?
2. How do we automatically identify the portion of data that is most useful to materialize?

In [7] we presented a framework for representing and using materialized data in an information agent. The basic idea is to locally materialize useful data and define it as another information source for the agent. The agent then considers using the materialized data instead of retrieving data from remote sources to answer user queries. For instance, in the countries application suppose we determined that the population and national product of all European countries was queried frequently and thus useful to materialize. The system would retrieve this data and define it as an additional information source as shown in Figure 15. Given a query the planner would use the materialized data instead of the original Web source(s) to answer the query when it reduced the cost of a query. The system selects the data to materialize by considering a combination of several factors which are described below.

6.1 The Distribution of User Queries

From the user query distribution we determine what classes of data are queried most frequently by users since it is often useful to materialize such classes. We have developed an algorithm known as the CM (Cluster and Merge) algorithm [8], which identifies useful classes of information to materialize by extracting patterns in user queries. A key feature of this algorithm is that it finds *compact* patterns to be extracted. Compact patterns are important from a query planning perspective since we define a new information source in the agent for each class of data materialized. A set of fragmented patterns will result in a large number of new sources. The problem of query planning in an information agent is combinatorially hard and having a very large number of sources could create performance problems for the planner. With compact patterns we need

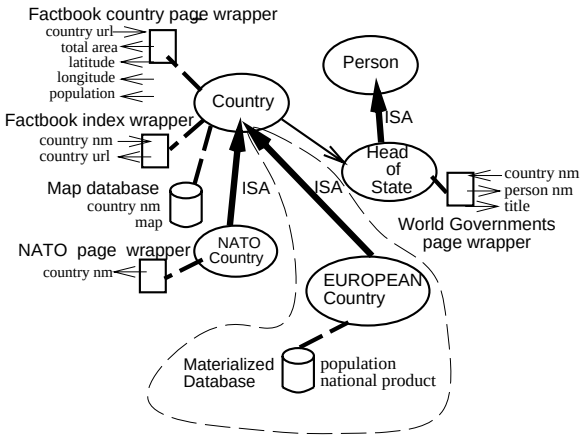


Figure 15: Materialized Data as Another Source

to materialize fewer classes of data and thus define fewer new information sources. For instance in the countries application suppose the class of information *"the population and ethnic divisions of all European countries"* was a frequently queried class of data, the CM algorithm would extract this class as a pattern from the query distribution. Even if some European countries were never queried, it might still materialize all of the data for the European countries in order to minimize the number of patterns stored.

6.2 The Structure of Queries and Sources

As described earlier, we provide database-like query access to otherwise only browsable Web sources by building wrappers around the sources. As a result certain kinds of queries can be very expensive as a lot of the functionality for structured querying of these sources needs to be provided by the wrapper or the mediator. For instance in the countries application, the CIA Factbook source is structured such that selection queries, such as say determining which countries are *European* countries requires the wrapper to retrieve pages of all countries in the Factbook source, which takes a long time.

In many kinds of queries we can materialize a portion of data which if stored locally greatly improves the response time of the expensive queries. In the above example if we materialize locally the names and continents of all countries, determining which countries are *European* countries (the step that requires otherwise fetching pages of all countries from the source) can be done locally and much faster. We have generalized this strategy for various kinds of queries that may be expensive such as selection queries, joins, ordered joins etc. The system analyzes the user query interface to determine what queries can be asked in an application and uses the mediator query cost estimator and also specification of the structure of Web sources to determine which of these queries can be expensive. It then prefetches and materializes data based on heuristics that use information about the kind of query (selection, join etc.) and the structure of the source (whether it supports selections etc.) to speed up the processing of the expensive queries.

6.3 Updates at Web Sources

Finally we must address the issue of updates at Web sources. First the materialized data must be kept consistent with that in the Web sources. It may also be that the user is willing to accept data that is not the most recent in exchange for a fast response to his query (using data that is materialized). Thus we need to determine the frequency with which each class of data materialized needs to be refreshed from the original sources. Next the total maintenance cost for all the classes of data materialized must be kept within a limit that can be handled by the system. We provide a language for describing the update characteristics of each source and also user requirements for freshness. The system then takes update frequency and maintenance cost into account when selecting data to materialize. For instance, considering the countries application again, the Factbook is a source that does not change so we materialize whatever data we want to prefetch or is frequently queried from that source as there is no maintenance cost for the materialized data. However we may not want to materialize data from the World Governments source as that is a source where data can change anytime.

We have implemented a materialization system for Ariadne based on the above ideas. We present experimental results in Table 1 demonstrating the effectiveness of our approach. Specifically we show the improvement in performance obtained by materializing data locally in Ariadne using our system.

Query Set	Response Time (No opt.)	Response Time (Ariadne)	Response Time (Page Level)	Improvement (Ariadne)	Improvement (Page Level)
Q1	41233 sec	1515 sec	36974 sec	96%	10%
Q2	47935 sec	2198 sec	42502 sec	95%	13%

Table 1: Experimental Results

Table 1 shows the experimental results for the countries application. We provide results showing the improvement in performance due to our system and compare it with an alternative scheme of page level caching with the same local space. We measure query response times with and without materialization for two query sets - Q1, which is a set of queries we generated and in which we introduced some distinct query patterns, and Q2, which is a set of actual user queries that we logged from a version of the countries application that was made available online for several months. There is significant performance improvement due to our materialization system. Our system also significantly outperforms the page level caching scheme for both query sets Q1 and Q2. The primary reason is that with our system we are much more flexible in selecting the portion of data that we want to materialize in terms of the attributes and tuples we want to specify. In the page level caching scheme, we can only materialize either all or none of an entire web page which causes local space to be wasted for storing data that may not be frequently queried. Also in this particular application source structure analysis proves to be very effective as the system locally materializes attributes of countries to speed the processing of certain very expensive kinds of selection queries.

7 Applications

We have used Ariadne to build a variety of applications. We list a few of them below to illustrate the generality of our approach.

As shown in the examples in this paper, we built a country information agent with Ariadne that combines data about the countries in the world from a variety of information sources. The agent extracts and integrates data from the CIA World Factbook. The Factbook is published each year, so the agent can support queries that span multiple years. The agent also extracts data from other related sources such as the World Governments site, which provides up-to-date information about world leaders. This application provides a good example where significant added value was added to the data sources by providing the additional structure where only the relevant data is extracted and returned for specific countries.

We have built a system using Ariadne that integrates data about restaurants and movie theaters and places the information on a map [9]. The application extracts restaurant names and addresses from CuisineNet, theater names and addresses from Yahoo Movies, movies and showtimes for each theater also come from Yahoo Movies, and trailers for movies from Hollywood.com. The addresses are run through the Etak geocoder to convert street addresses into the corresponding latitude and longitude. Finally, Ariadne uses the US Census Bureau's Tiger Map Server to produce a map with all of the restaurants and theaters for a requested city. Clicking on one of the points of the map will bring up the restaurant review or list of movies as appropriate. If the user selects a specific movie, Ariadne will then play the appropriate trailer. This application provides a compelling integration example that combines a variety of multimedia data sources.

We have applied Ariadne to integrate online electronic catalogs to provide a more comprehensive virtual catalog. In this application, Ariadne combines data on pricing, availability, manufacturer, and so on from four major electronic part suppliers. Each site requires several wrappers in order to navigate to the page that provides the required data. In several cases, the data on a given part is spread over multiple pages at a site. This application provides an example of a virtual data source where up-to-date information on electronic parts is made available from multiple suppliers without maintaining any local data.

8 Discussion

There is a large body of literature on information integration [30], and more recently, several projects focusing specifically on information integration on the Web, such as Tukwila [29], Araneus [25], the Information Manifold [23], Occam [22], Infomaster [15], and InfoSleuth [17]. These systems focus on a variety of issues, including the problems of representing and selecting a relevant set of sources to answer a query, handling binding patterns, and resolving discrepancies among sources. All of this work is directly relevant to Ariadne, but no other system handles the broad range of practical issues that arise in modeling information within a single page, across pages at a site, and across sites to support web-based information integration.

We believe that Ariadne is successful, in terms of the broad applicability of the approach, because it combines a simple representation scheme with sophisticated modeling tools that map web information sources into this simple representation. There are

many examples of impressive AI systems based on relatively simple representational schemes. In the realm of planning, recent examples include SATplan [18] and Graphplan [10]; the former employs a propositional CSP approach, the latter, a graph-based search. In machine learning, propositional learning schemes (e.g., decision trees) have been dominant. Though it is often difficult to understand exactly what a simple representational scheme buys you computationally, one thing seems clear: systems with simple representations are often easier to design and understand.

Ariadne's representation scheme was adopted from database systems, where the world consists of a set of relations (or tables) over objects, and simple relational operators (retrieve, join, etc.) are composed to answer queries. This representation makes it straightforward to integrate multiple databases using an AI planner. Ariadne's planner can efficiently search for a sequence of joins, selections, etc. that will produce the desired result without needing to do any sophisticated reasoning about the information sources themselves.

The Web environment is much more than a set of relational tables, of course. What makes Ariadne possible are the modeling tools that enable a user to create a database-like view of the Web. Other competing approaches to information integration on the web (such as the ones mentioned above) have also adopted database-oriented representational schemes, but they do not include the tools that enable developers to create these models for real web sites. Where our approach becomes challenging (and could break down) is in situations where the "natural" way to represent a web site is not possible due to limitations of the underlying representation.

One such limitation is that Ariadne cannot reason about recursive relations. (To do this properly would require query plans to contain loops.) This has many practical ramifications. For example, consider web pages that have a 'more' button at the bottom, such as Alta Vista's response pages. It would be natural to represent each 'more' button as a pointer to the next page in a list, but there is no way to do this without a recursive relation. Instead, we can build knowledge about 'more' buttons in our wrapper generation tools, so the process of following a 'more' buttons is done completely within a wrapper, hiding the complexity from the query planner.

Another ramification of the planner's inability to reason about recursive relations shows up with hierarchical indexes like Yahoo, where there is no fixed depth to the hierarchy. The natural way to model such pages is with a parent-child relation. Instead, the alternative is to build a more sophisticated wrapper that computes the transitive closure of the parent-child relationship, so that we can obtain all of a node's descendants in one step.

There is an obvious tension between the expressiveness of the representation and the burden we place on the modeling tools. Some researchers, such as Friedman et al. [14], have introduced more expressive representations for modeling web pages and their interconnections. Our approach has been to keep the representation and planning process simple, compensating for their limitations by relying on smarter modeling tools. As we have described, the advantage is that we can incrementally build a suite of modeling tools that use machine learning, statistical inference, and other AI techniques, producing a system that can handle a surprisingly wide range of tasks.

9 Acknowledgements

The research reported here was supported in part by the Rome Laboratory of the Air Force Systems Command and the Defense Advanced Research Projects Agency (DARPA) under contract number F30602-98-2-0109, in part by the United States Air Force under contract number F49620-98-1-0046, and in part by the Integrated Media Systems Center, a National Science Foundation Engineering Research Center, Cooperative Agreement No. EEC-9529152. The views and conclusions contained in this article are the authors' and should not be interpreted as representing the official opinion or policy of any of the above organizations or any person connected with them.

References

[1] José Luis Ambite. *Planning by Rewriting.* PhD thesis, Department of Computer Science, University of Southern California, 1998.

[2] José Luis Ambite and Craig A. Knoblock. Planning by rewriting: Efficiently generating high-quality plans. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, Providence, RI, 1997.

[3] José Luis Ambite and Craig A. Knoblock. Flexible and scalable query planning in distributed and heterogeneous environments. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, Pittsburgh, PA, 1998.

[4] José Luis Ambite and Craig A. Knoblock. Flexible and scalable cost-based query planning in mediators: A transformational approach. *Artificial Intelligence Journal*, 118(1-2):115–161, April 2000.

[5] José Luis Ambite, Craig A. Knoblock, Ion Muslea, and Andrew Philpot. Compiling source descriptions for efficient and flexible information integration. To appear in Journal of Intelligent Information Systems, 2000.

[6] Yigal Arens, Craig A. Knoblock, and Wei-Min Shen. Query reformulation for dynamic information integration. *Journal of Intelligent Information Systems, Special Issue on Intelligent Information Integration*, 6(2/3):99–130, 1996.

[7] Naveen Ashish, Craig A. Knoblock, and Cyrus Shahabi. Intelligent caching for information mediators: A kr based approach. In *Knowledge Representation meets Databases (KRDB)*, Seattle, WA, 1998.

[8] Naveen Ashish, Craig A. Knoblock, and Cyrus Shahabi. Selectively materializing data in mediators by analyzing user queries. In *Fourth International Conference on Cooperative Information Systems (CoopIS)*, Edinburgh, Scotland, September 1999.

[9] Greg Barish, Craig A. Knoblock, Yi-Shin Chen, Steven Minton, Andrew Philpot, and Cyrus Shahabi. The TheaterLoc virtual application. In *Proceedings of Twelfth Annual Conference on Innovative Applications of Artificial Intelligence (IAAI-2000)*, Austin, Texas, 2000.

[10] Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1636–1642, August 1995.

[11] M. Califf and R. Mooney. Relational learning of pattern-match rules for information extraction. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 328–334, 1999.

[12] William W. Cohen. Knowledge integration for structured information sources containing text (extended abstract). In *SIGIR-97 Workshop on Networked Information Retrieval*, 1997.

[13] D. Freitag. Information extraction from HTML: Application of a general learning approach. In *Proceedings of the 15th Conference on Artificial Intelligence (AAAI-98)*, pages 517–523, 1998.

[14] Marc Friedman, Alon Levy, and Todd Millstein. Navigational plans for data integration. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 67–73, Orlando, Florida, USA, August 1999. AAAI Press / The MIT Press.

[15] Michael R. Genesereth, Arthur M. Keller, and Oliver M. Duschka. Infomaster: An information integration system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, AZ, 1997.

[16] C. Hsu and M. Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Journal of Information Systems*, 23(8):521–538, 1998.

[17] R.J. Bayardo Jr., W. Bohrer, R. Brice, A. Cichocki, J. Fowler, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk. Infosleuth: Agent-based semantic integration of information in open and dynamic environments. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, AZ, 1997.

[18] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1202–1207, Portland, Oregon, USA, August 1996. AAAI Press / The MIT Press.

[19] Craig A. Knoblock. Planning, executing, sensing, and replanning for information gathering. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, Montreal, Canada, 1995.

[20] Craig A. Knoblock, Steven Minton, José Luis Ambite, Naveen Ashish, Pragnesh Jay Modi, Ion Muslea, Andrew G. Philpot, and Sheila Tejada. Modeling web sources for information integration. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, Madison, WI, 1998.

[21] Nicholas Kushmerick. *Wrapper Induction for Information Extraction*. PhD thesis, Department of Computer Science and Engineering, University of Washington, 1997.

[22] Chung T. Kwok and Daniel S. Weld. Planning to gather information. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR, 1996.

[23] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Query-answering algorithms for information agents. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR, 1996.

[24] Robert MacGregor. A deductive pattern matcher. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, Saint Paul, Minnesota, 1988.

[25] G. Mecca, P. Atzeni, A. Masci, P. Merialdo, and G. Sindoni. From databases to web-bases: The araneus experience. Technical Report 34-1998, Dipartimento di Informatica e Automazione, Universita' di Roma Tre, May 1998.

[26] Ion Muslea, Steven Minton, and Craig Knoblock. Hierarchical wrapper induction for semistructured information sources. *Journal of Autonomous Agents and Multi-Agent Systems*, Forthcoming.

[27] S. Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1/2/3):233–272, 1999.

[28] Jeffrey D. Ullman. Information integration using logical views. In *Proceedings of the Sixth International Conference on Database Theory*, Delphi, Greece, January 1997.

[29] Zachary G. Ives Daniela Florescu Marc Friedman Alon Levy Daniel S. Weld. An adaptive query execution system for data integration. In *Proceedings of 1999 ACM SIGMOD*, Philadelphia, PA, 1999.

[30] Gio Wiederhold. *Intelligent Integration of Information.* Kluwer, 1996.

Appendix D:

Accurately and Reliably Extracting Data from the Web: A Machine Learning Approach*†

Craig A. Knoblock
University of Southern California and Fetch Technologies

Kristina Lerman
University of Southern California

Steven Minton
Fetch Technologies

Ion Muslea
University of Southern California

Abstract

A critical problem in developing information agents for the Web is accessing data that is formatted for human use. We have developed a set of tools for extracting data from web sites and transforming it into a structured data format, such as XML. The resulting data can then be used to build new applications without having to deal with unstructured data. The advantages of our wrapping technology over previous work are the the ability to learn highly accurate extraction rules, to verify the wrapper to ensure that the correct data continues to be extracted, and to automatically adapt to changes in the sites from which the data is being extracted.

1 Introduction

There is a tremendous amount of information available on the Web, but much of this information is not in a form that can be easily used by other applications. There are hopes that XML will solve this problem, but XML is not yet in widespread use and even in the

*Appeared in IEEE Data Engineering Bulletin, 23(4), December, 2000.

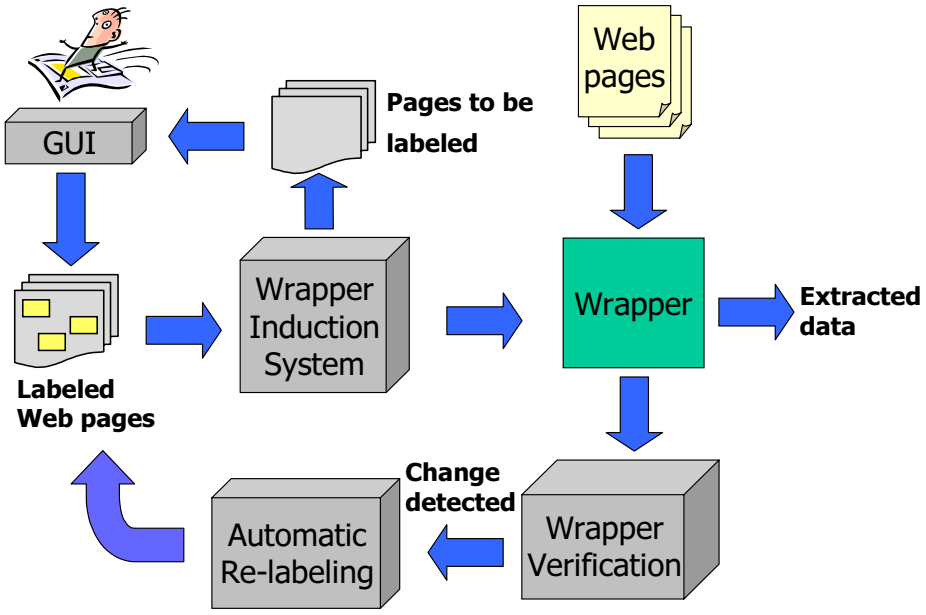


Figure 1: **The Lifecycle of a Wrapper**

best case it will only address the problem within application domains where the interested parties can agree on the XML schema definitions. Previous work on wrapper generation in both academic research [4, 6, 8] and commercial products (such as OnDisplay's eContent) have primarily focused on the ability to rapidly create wrappers. The previous work makes no attempt to ensure the accuracy of the wrappers over the entire set of pages of a site and provides no capability to detect failures and repair the wrappers when the underlying sources change.

We have developed the technology for rapidly building wrappers for accurately and reliably extracting data from semistructured sources. Figure 1 graphically illustrates the entire lifecycle of a wrapper. As shown in the Figure, the wrapper induction system takes a set of web pages labeled with examples of the data to be extracted. The user provides the initial set of labeled examples and the system can suggest additional pages for the user to label in order to build wrappers that are very accurate. The output of the wrapper induction system is a set of extraction rules that describe how to locate the desired information on a Web page. After the system creates a wrapper, the wrapper verification system uses the functioning wrapper to learn patterns that describe the data being extracted. If a change is detected, the system can automatically repair a wrapper by using the same patterns to locate examples on the changed pages and re-running the wrapper induction system. The details of this entire process are described in the remainder of this paper.

2 Learning Extraction Rules

A wrapper is a piece of software that enables a semistructured Web source to be queried as if it were a database. These are sources where there is no explicit structure or schema,

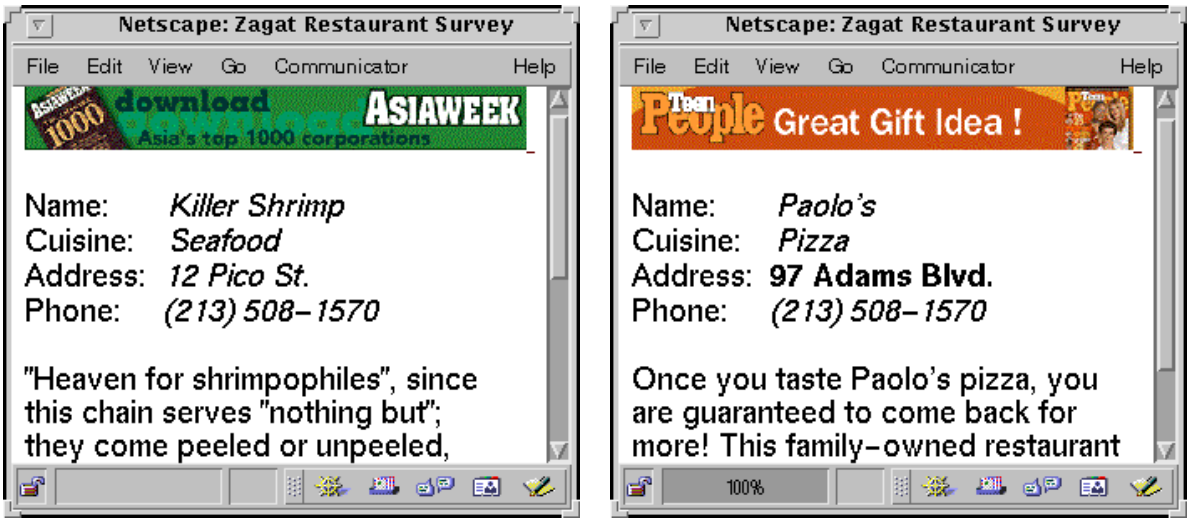


Figure 2: **Two Sample Restaurant Documents From the Zagat Guide.**

but there is an implicit underlying structure (for example, consider the two documents in Figure 2). Even text sources, such as email messages, have some structure in the heading that can be exploited to extract the date, sender, addressee, title, and body of the messages. Other sources, such as online catalogs, have a very regular structure that can be exploited to extract the data automatically.

One of the critical problems in building a wrapper is defining a set of extraction rules that precisely define how to locate the information on the page. For any given item to be extracted from a page, one needs an extraction rule to locate both the beginning and end of that item. Since, in our framework, each document consists of a sequence of tokens (e.g., words, numbers, HTML tags, etc), this is equivalent to finding the first and last tokens of an item. The hard part of this problem is constructing a set of extraction rules that work for *all* of the pages in the source.

A key idea underlying our work is that the extraction rules are based on "landmarks" (i.e., groups of consecutive tokens) that enable a wrapper to locate the start and end of the item within the page. For example, let us consider the three restaurant descriptions `E1`, `E2`, and `E3` presented in Figure 3. In order to identify the beginning of the address, we can use the rule

```
E1:   ...Cuisine:<i>Seafood</i><p>Address:<i>  12 Pico St. </i><p>Phone:<i>...
E2:   ...Cuisine:<i>Thai   </i><p>Address:<i> 512 Oak Blvd.</i><p>Phone:<i>...
E3:   ...Cuisine:<i>Burgers</i><p>Address:<i> 416 Main St. </i><p>Phone:<i>...
E4:   ...Cuisine:<i>Pizza</i><p>Address:<b> 97 Adams Blvd. </b><p>Phone:<i>...
```

Figure 3: **Four sample restaurant documents.**

R1 = *SkipTo*(`Address`) *SkipTo*(`<i>`)

which has the following meaning: start from the beginning of the document and skip every token until you find a landmark consisting of the word `Address`, and then, again, ignore everything until you find the landmark `<i>`. **R1** is called a *start rule* because it identifies the beginning of the address. One can write a similar *end rule* that finds the end of the address; for sake of simplicity, we restrict our discussion here to start rules.

Note that **R1** is by no means the only way to identfy the beginning of the address. For instance, the rules

R2 = *SkipTo*(`Address : <i>`)

R3 = *SkipTo*(`Cuisine : <i>`) *SkipTo*(`Address : <i>`)

R4 = *SkipTo*(`Cuisine : <i>` *_Capitalized_* `</i> <p> Address : <i>`)

can be also used as start rules. **R2** uses the 3-token landmark that immediately precedes the beginning of the address in examples `E1`, `E2`, and `E3`, while **R3** relies on two 3-token landmarks. Finally, **R4** is defined based on a 9-token landmark that uses the wildcard *_Capitalized_*, which is a placeholder for any capitalized alphabetic string (other examples of useful wildcards are *_Number_*, *_AllCaps_*, *_HtmlTag_*, etc).

To deal with variations in the format of the documents, our extraction rules allow the use of *disjunctions*. For example, let us assume that the addresses that are within one mile from your location appear in bold (see example `E4` in Figure 3), while the other ones are displayed as italic (e.g., `E1`, `E2`, and `E3`). We can extract all the names based on the disjunctive start rule

either *SkipTo*(`Address : <b>`)

or *SkipTo*(`Address`) *SkipTo*(`<i>`)

Disjunctive rules are *ordered lists* of individual disjuncts (i.e., decision lists). Applying a disjunctive rule is a straightforward process: the wrapper successively applies each disjunct in the list until it finds the first one that matches. Even though in our simplified examples one could have used the nondisjunctive rule

SkipTo(`Address :` *_HtmlTag_*),

there are many real world sources that cannot be wrapped without using disjuncts.

We have developed STALKER [11], a *hierarchical* wrapper induction algorithm that learns extraction rules based on examples labeled by the user. We have a graphical user interface that allows a user to mark up several pages on a site, and the system then generates a set of extraction rules that accurately extract the required information. Our approach uses a greedy-covering inductive learning algorithm, which incrementally builds the extraction rules from the examples.

In contrast to other approaches [4, 6, 8], a key feature of STALKER is that it is able to efficiently generate extraction rules from a small number of examples: it rarely requires more than 10 examples, and in many cases two examples are sufficient. The ability to generalize from such a small number of examples has a two-fold explanation. First, in most of the

cases, the pages in a source are generated based on a fixed template that may have only a few variations. As STALKER tries to learn landmarks that are part of this template, it follows that for templates with little or no variations a handful of examples usually will be sufficient to induce reliable landmarks.

Second, STALKER exploits the *hierarchical* structure of the source to constrain the learning problem. More precisely, based on the schema of the data to be extracted, we *automatically* decompose one difficult problem (i.e., extract all items of interest) into a series of simpler ones. For instance, instead of using one complex rule that extracts all restaurant names, addresses and phone numbers from a page, we take a hierarchical approach. First we apply a rule that extracts the whole list of restaurants; then we use another rule to break the list into tuples that correspond to individual restaurants; finally, from each such tuple we extract the name, address, and phone number of the corresponding restaurant. Our hierarchical approach also has the advantage of being able to extract data from pages that contain complicated formatting layouts (e.g., lists embedded in other lists) that previous approaches could not handle (see [11] for details).

STALKER is a sequential covering algorithm that, given the training examples E, tries to learn a minimal number of *perfect disjuncts* that cover *all* examples in E. By definition, a perfect disjunct is a rule that covers at least one training example and on any example the rule matches it produces the correct result. STALKER first creates an initial set of candidate-rules C and then repeatedly applies the following three steps until it generates a perfect disjunct:

- select most promising candidate from C
- refine that candidate
- add the resulting refinements to C

Once STALKER obtains a perfect disjunct P, it removes from E all examples on which P is correct, and the whole process is repeated until there are no more training examples in E. STALKER uses two types of refinements: *landmark refinements* and *topology refinements*. The former makes the rule more specific by adding a token to one of the existing landmarks, while the latter adds a new 1-token landmark to the rule.

For instance, let us assume that based on the four examples in Figure 3, we want to learn a start rule for the address. STALKER proceeds as follows. First, it selects an example, say `E4`, to guide the search. Second, it generates a set of *initial candidates*, which are rules that consist of a single 1-token landmark; these landmarks are chosen so that they match the token that *immediately precedes* the beginning of the address in the guiding example. In our case we have two initial candidates:

R5 = *SkipTo*(<b>)

R6 = *SkipTo*(*_HtmlTag_*)

As the <b> token appears only in `E4`, **R5** does not match within the other three examples. On the other hand, **R6** matches in all four examples, even though it matches *too early* (**R6** stops as soon as it encounters an HTML tag, which happens in all four examples *before*

the beginning of the address). Because **R6** has a better generalization potential, STALKER selects **R6** for further refinements.

While refining **R6**, STALKER creates, among others, the new candidates **R7**, **R8**, **R9**, and **R10** shown below. The first two are obtained via landmark refinements (i.e., a token is added to the landmark in **R6**), while the other two rules are created by topology refinements (i.e., a new landmark is added to **R6**). As **R10** works correctly on all four examples, STALKER stops the learning process and returns **R10**.

R7 = *SkipTo*(: *_HtmlTag_*)

R8 = *SkipTo*(*_Punctuation_* *_HtmlTag_*)

R9 = *SkipTo*(:) *SkipTo*(*_HtmlTag_*)

R10 = *SkipTo*(`Address`) *SkipTo*(*_HtmlTag_*)

By using STALKER, we were able to successfully wrap information sources that could not be wrapped with existing approaches (see [11] for details). In an empirical evaluation on 28 sources proposed in [8], STALKER had to learn 206 extraction rules. We learned 182 *perfect* rules (100% accurate), and another 18 rules that had an accuracy of at least 90%. In other words, only 3% of the learned rules were less that 90% accurate .

3 Identifying Highly Informative Examples

STALKER can do significantly better on the hard tasks (i.e., the ones for which it failed to learn perfect rules) if instead of *random* examples, the system is provided with carefully selected examples. Specifically, the most informative examples illustrate exceptional cases. However, it is unrealistic to assume that a user is willing and has the skills to browse a large number of documents in order to identify a sufficient set of examples to learn perfect extraction rules. This is a general problem that none of the existing tools address, regardless of whether they use machine learning.

To solve this problem we have developed an *active learning* approach that analyzes the set of unlabeled examples to automatically select examples for the user to label. Our approach, called *co-testing* [10], exploits the fact that there are often multiple ways of extracting the same information [1]. In the case of wrapper learning, the system can learn two different types of rules: *forward* and *backward* rules. All the rules presented above are *forward* rules: they start at the beginning of the document and go towards the end. By contrast, a *backward* rule starts at the end of the page and goes towards its beginning. For example, one can find the beginning of the addresses in Figure 3 by using one of the following backward rules:

R11 = *BackTo*(`Phone`) *BackTo*(*_Number_*)

R12 = *BackTo*(`Phone : <i>`) *BackTo*(*_Number_*)

The main idea behind co-testing is straightforward: after the user labels one or two examples, the system learns *both* a forward and a backward rule. Then it runs *both* rules on a given set of unlabeled pages. Whenever the rules disagree on an example, the system considers that as an example for the user to label next. The intuition behind our approach is the following: if both rules are 100% accurate, on *every* page they must identify *the same* token as the beginning of the address. Furthermore, as the two rules are learned based on different sets of tokens (i.e., the sequences of tokens that precede and follow the beginning of the address, respectively), they are highly unlikely to make the exact same mistakes. Whenever the two rules disagree, at least one of them must be wrong, and by asking the user to label that particular example, we obtain a highly informative training example. Co-testing makes it possible to generate accurate extraction rules with a very small number of labeled examples.

To illustrate how co-testing works, consider again the examples in Figure 3. Since most of the restaurants in a city are *not* located within a 1-mile radius of one's location, it follows that most of the documents in the source will be similar to `E1`, `E2`, and `E3` (i.e., addresses shown in italic), while just a few examples will be similar to `E4` (i.e., addresses shown in bold). Consequently, it is unlikely that an address in bold will be present in a small, randomly chosen, initial training set. Let us now assume that the initial training set consists of `E1` and `E2`, while `E3` and `E4` are *not labeled*. Based on these examples, we learn the rules

Fwd-R1 = $SkipTo($`Address`$)$ $SkipTo($`<i>`$)$

Bwd-R1 = $BackTo($`Phone`$)$ $BackTo($`_Number_`$)$

Both rules correctly identify the beginning of the address for all restaurants that are more than one mile away, and, consequently, they will agree on all of them (e.g., `E3`). On the other hand, **Fwd-R1** works *incorrectly* for examples like `E4`, where it stops at the beginning of the phone number. As **Bwd-R1** is correct on `E4`, the two rules disagree on this example, and the user is asked to label it.

To our knowledge, there is no other wrapper induction algorithm that has the capability of identifying the most informative examples. In the related field of information extraction, where one wants to extract data from free text documents, researchers proposed such algorithms [13, 12], but they cannot be applied in a straightforward manner to existing wrapper induction algorithms.

We applied co-testing on the 24 tasks on which STALKER fails to learn perfect rules based on 10 random examples. To keep the comparison fair, co-testing started with one random example and made up to 9 queries. The results were excellent: the average accuracy over all tasks improved from 85.7% to 94.2% (error rate reduced by 59.5%). Furthermore, 10 of the learned rules were 100% accurate, while another 11 rules were at least 90% accurate. In these experiments as well as in other related tests [10] applying co-testing leads to a significant improvement in accuracy without having to label more training data.

4 Verifying the Extracted Data

Another problem that has been largely ignored in past work on extracting data from web sites is that sites change and they change often. Kushmerick [7] addressed the wrapper verification problem by monitoring a set of generic features, such as the density of numeric characters within a field, but this approach only detects certain types of changes. In contrast, we address this problem by applying machine learning techniques to learn a set of patterns that describe the information that is being extracted from each of the relevant fields. Since the information for even a single field can vary considerably, the system learns the statistical distribution of the patterns for each field. Wrappers can be verified by comparing the patterns of data returned to the learned statistical distribution. When a significant difference is found, an operator can be notified or we can automatically launch the wrapper repair process, which is described in the next section.

The learned patterns represent the structure, or format, of the field as a sequence of words and wildcards [9]. Wildcards represent syntactic categories to which words belong — alphabetic, numeric, capitalized, etc. — and allow for multi-level generalization. For complex fields, and for purposes of information extraction, it is sufficient to use only the starting and ending patterns as the description of the data field. For example, a set of street addresses — **12 Pico St.**, **512 Oak Blvd.**, **416 Main St.** and **97 Adams Blvd.** — all start with a pattern (*_Number_ _Capitalized_*) and end with **(Blvd.)** or **(St.)**. We refer to the starting and ending patterns together as the *data prototype* of the field.

The problem of learning a description of a field (class) from a set of labeled examples can be posed in two related ways: as a classification or a conservation task. If negative examples are present, the classification algorithm learns a *discriminating* description. When only positive examples are available, the conservation task learns a *characteristic* description, *i.e.* one that describes many of the positive examples but is highly unlikely to describe a randomly generated example. Because an appropriate set of negative examples not available in our case, we chose to frame the learning problem as a conservation task. The algorithm we developed, called DataPro [9], finds all statistically significant starting and ending patterns in a set of positive examples of the field. A pattern is said to be significant if it occurs more frequently than would be expected by chance if the tokens were generated randomly and independently of one another. Our approach is similar to work on grammar induction [2, 5], but our pattern language is better suited to capturing the regularities in small data fields (as opposed to languages).

The algorithm operates by building a prefix tree, where each node corresponds to a token whose position in the sequence is given by the node's depth in the tree. Every path through the tree starting at the root node is a significant pattern found by the algorithm.

The algorithm grows the tree incrementally. Adding a child to a node corresponds to extending the node's pattern by a single token. Thus, each child represents a different way to specialize the pattern. For example, when learning city names, the node corresponding to "New" might have three children, corresponding to the patterns "New Haven", "New York" and "New *_Capitalized_*". A child node is judged to be significant with respect to its parent node if the number of occurrences of the child pattern is sufficiently large, given the number of occurrences of the parent pattern and the baseline probability of the token used

to extend the parent pattern. A pruning step insures that each child node is also significant given its more specific siblings. For example, if there are 10 occurrence of "New Haven", and 12 occurrences of "New York", and both of these are judged to be significant, then "New _*Capitalized*_" will be retained only if there are significantly more than 22 examples that match "New _*Capitalized*_". Similarly, once the entire tree has been expanded, the algorithm includes a pattern extraction step that traverses the tree, checking whether the pattern "New" is still significant given the more specific patterns "New York", "New Haven" and "New _*Capitalized*_". In other words, DataPro decides whether the examples described by "New" but not by any of the longer sequences can be explained by the null hypothesis.

The patterns learned by DataPro lend themselves to the data validation task and, specifically, to wrapper verification. A set of queries is used to retrieve HTML pages from which the wrapper extracts some training examples. The learning algorithm finds the patterns that describe the common beginnings and endings of each field of the training examples. In the verification phase, the wrapper generates a test set of examples from pages retrieved using the same or similar set of queries. If the patterns describe statistically the same (at a given significance level) proportion of the test examples as the training examples, the wrapper is judged to be extracting correctly; otherwise, it is judged to have failed.

Once we have learned the patterns, which represent our expectations about the format of data, we can then configure the wrappers to verify the extracted data before the data is sent, immediately after the results are sent, or on some regular frequency. The most appropriate verification method depends on the particular application and the tradeoff between response time and data accuracy.

We monitored 27 wrappers (representing 23 distinct Web sources) over a period of several months. For each wrapper, the results of 15-30 queries were stored periodically. All new results were compared with the last correct wrapper output (training examples). A manual check of the results revealed 37 wrapper changes out of the total 443 comparisons. The verification algorithm correctly discovered 35 of these changes. The algorithm incorrectly decided that the wrapper has changed in 40 cases. We are currently working to reduce the high rate of false positives.

5 Automatically Repairing Wrappers

Most changes to Web sites are largely syntactic and are often minor formatting changes or slight reorganizations of a page. Since the content of the fields tend to remain the same, we exploit the patterns learned for verifying the extracted results to locate correct examples of the data field on new pages. Once the required information has been located, the pages are automatically re-labeled and the labeled examples are re-run through the inductive learning process to produce the correct rules for this site.

Specifically, the wrapper reinduction algorithm takes a set of training examples and a set of pages from the same source and uses machine learning techniques to identify examples of the data field on new pages. First, it learns the starting and ending patterns that describe each field of the training examples. Next, each new page is scanned to identify all text segments that begin with one of the starting patterns and end with one of the ending patterns. Those segments, which we call candidates, that are significantly longer or shorter than the

Figure 4: **An Example of the Reinduction Process**

training examples are eliminated from the set, often still leaving hundreds of candidates per page. The candidates are then clustered to identify subgroups that share common features. The features used in clustering are the candidate's relative position on the page, adjacent landmarks, and whether it is visible to the user. Each group is then given a score based on how similar it is to the training examples. We expect the highest ranked group to contain the correct examples of the data field.

Figure 4 shows a actual example of a change to Amazon's site and how our system automatically adapts to the change. The top frame shows an example of the original site, the extraction rule for a book title, and the extracted results from the example page. The middle frame shows the source and the incorrectly extracted result after the title's font and color were changed. And the bottom frame shows the result of the automatic reinduction process with the corrected rule for extracting the title.

We evaluated the algorithm by using it to extract data from Web pages for which correct output is known. The output of the extraction algorithm is a ranked list of clusters for every data field being extracted. Each cluster is checked manually, and it is judged to be correct

if it contains only the complete instances of the field, which appear in the correct context on the page. For example, if extracting a city of an address, we only want to extract those city names that are part of an address.

We ran the extraction algorithm for 21 distinct Web sources, attempting to extract 77 data fields from all the sources. In 62 cases the top ranked cluster contained correct complete instances of the data field. In eight cases the correct cluster was ranked lower, while in six cases no candidates were identified on the pages. The most closely related work is that of Cohen [3], which uses an information retrieval approach to relocating the information on a page. The approach was not evaluated on actual changes to Web pages, so it is difficult to assess whether this approach would work in practice.

6 Discussion

Our work addresses the complete wrapper creation problem, which includes:

- building wrappers by example,
- ensuring that the wrappers accurately extract data across an entire collection of pages,
- verifying a wrapper to avoid failures when a site changes,
- and automatically repair wrappers in response to changes in layout or format.

Our main technical contribution is in the use of machine learning to accomplish all of these tasks. Essentially, our approach takes advantage of the fact that web pages have a high degree of regular structure. By analyzing the regular structure of example pages, our wrapper induction process can detect landmarks that enable us to extract desired fields. After we developed an initial wrapper induction process we realized that the accuracy of the induction method can be improved by simultaneously learning "forward" and "backward" extraction rules to identify exception cases. Again, what makes this possible is the regularities on a page that enable us to identify landmarks both before a field and after a field.

Our approach to automatically detecting wrapper breakages and repairing them capitalizes on the regular structure of the extracted fields themselves. Once a wrapper has been initially created, we can use it to obtain numerous examples of the fields. This enables us to profile the information in the fields and obtain structural descriptions that we can use during monitoring and repair. Essentially this is a bootstrapping approach. Given the initial examples provided by the user, we first learn a wrapper. Then we use this wrapper to obtain many more examples which we then analyze in much greater depth. Thus by leveraging a few human-provided examples, we end up with a highly scalable system for wrapper creation and maintenance.

Acknowledgements

The research reported here was supported in part by the Rome Laboratory of the Air Force Systems Command and the Defense Advanced Research Projects Agency (DARPA) under

contract number F30602-98-2-0109, in part by the United States Air Force under contract number F49620-98-1-0046, and in part by the Integrated Media Systems Center, a National Science Foundation Engineering Research Center, under Cooperative Agreement No. EEC-9529152. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

References

[1] A. Blum and T. Mitchell. Combining labeled and unlabeled data with co-training. In *Proc. of the 1988 Conference on Computational Learning Theory*, pages 92–100, 1998.

[2] R. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In *Lecture Notes In Computer Science*, page 862, 1994.

[3] W. Cohen. Recognizing structure in web pages using similarity queries. In *Proc. of the 16th National Conference on Artificial Intelligence AAAI-1999*, pages 59–66, 1999.

[4] D. Freitag and N. Kushmerick. Boosted wrapper induction. In *Proc. of the 17th National Conference on Artificial Intelligence AAAI-2000*, pages 577–583, 2000.

[5] T. Goan, N. Benson, and O. Etzioni. A grammar inference algorithm for the world wide web. In *Proc. of the AAAI Spring Symposium on Machine Learning in Information Access*, 1996.

[6] C. Hsu and M. Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Journal of Information Systems*, 23(8):521–538, 1998.

[7] N. Kushmerick. Regression testing for wrapper maintenance. In *Proc. of the 16th National Conference on Artificial Intelligence AAAI-1999*, pages 74–79, 1999.

[8] N. Kushmerick. Wrapper induction: efficiency and expressiveness. *Artificial Intelligence Journal*, 118(1-2):15–68, 2000.

[9] K. Lerman and S. Minton. Learning the common structure of data. In *Proc. of the 17th National Conference on Artificial Intelligence AAAI-2000*, pages 609–614, 2000.

[10] I. Muslea, S. Minton, and C. Knoblock. Co-testing: Selective sampling with redundant views. In *Proc. of the 17th National Conference on Artificial Intelligence AAAI-2000*, pages 621–626, 2000.

[11] I. Muslea, S. Minton, and C. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Journal of Autonomous Agents and Multi-Agent Systems*, 2001. (to appear).

[12] S. Soderland. Learning extraction rules for semi-structured and free text. *Machine Learning*, 34:233–272, 1999.

[13] C. Thompson, M. Califf, and R. Mooney. Active learning for natural language parsing and information extraction. In *Proc. of the 16th International Conference on Machine Learning ICML-99*, pages 406–414, 1999.

Appendix E:

Wrapper Maintenance: A Machine Learning Approach *

Kristina Lerman[1], Steven Minton[2], Craig Knoblock[1,2]
1. Information Sciences Institute
Univ. of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
2. Fetch Technologies
4676 Admiralty Way
Marina del Rey, CA 90292
{lerman,knoblock}@isi.edu
minton@fetch.com

Abstract

The proliferation of online information sources has led to an increased use of wrappers for extracting data from Web sources. While most of the previous research has focused on quick and efficient generation of wrappers, the development of tools for wrapper maintenance has received less attention. This is an important research problem because Web sources often change in ways that prevent the wrappers from extracting data correctly. We present an efficient algorithm that learns structural information about data from positive examples alone. We describe how this information can be used for two wrapper maintenance applications: wrapper verification and reinduction. The wrapper verification system detects when a wrapper is not extracting correct data, usually because the Web source has changed its format. The reinduction algorithm automatically recovers from changes in the Web source by identifying data on Web pages so that a new wrapper may be generated for this source. To validate our approach, we monitored 27 wrappers over a period of a year. The verification algorithm correctly discovered 35 of the 37 wrapper changes, and made 16 mistakes, resulting in precision of 0.73 and recall of 0.95. We validated the reinduction algorithm on ten Web sources. We were able to successfully reinduce the wrappers, obtaining precision and recall values of 0.90 and 0.80 on the data extraction task. Our results indicate that combining structural information about data with *a priori* expectations about page structure is a feasible approach to automatically generating wrappers.

*Submitted to *Journal of Artificial Intelligence Research*

1 Introduction

There is a tremendous amount of information available online, but much of this information is formatted to be easily read by human users, not computer applications. Extracting information from semi-structured Web pages is an increasingly important capability for Web-based software applications that perform information management functions, such as shopping agents [Doorenbos *et al.*, 1997] and virtual travel assistants [Knoblock *et al.*, 2001b, Ambite *et al.*, 2002], among others. These applications, often referred to as agents, rely on Web wrappers that extract information from semi-structured sources and convert it to a structured format. Semi-structured sources are those that have no explicitly specified grammar or schema, but have an implicit grammar that can be used to identify relevant information on the page. Even text sources such as email messages have some structure in the heading that can be exploited to extract the date, sender, addressee, title, and body of the messages. Other sources, such as online catalogs, have a very regular structure that can be exploited to extract all the data automatically.

Wrappers rely on extraction rules to identify the data field to be extracted. Semi-automatic creation of extraction rules, or wrapper induction, has been an active area of research in recent years [Knoblock *et al.*, 2001a, Kushmerick *et al.*, 1997]. The most advanced of these wrapper generation systems use machine learning techniques to learn the extraction rules by example. For instance, the wrapper induction tool developed at USC [Knoblock *et al.*, 2001a, Muslea *et al.*, 1998] and commercialized by Fetch Technologies, allows the user to mark up data to be extracted on several example pages from an online source using a graphical user interface. The system then generates "landmark"-based extraction rules for these data that rely on the page layout. The USC wrapper tool is able to efficiently create extraction rules from a small number of examples; moreover, it can extract data from pages that contain lists, nested structures, and other complicated formatting layouts.

In comparison to wrapper induction, wrapper maintenance has received less attention. This is an important problem, because even slight changes in the Web page layout can break a wrapper that uses landmark-based rules and prevent it from extracting data correctly. In this paper we discuss our approach to the wrapper maintenance problem, which consists of two parts: wrapper verification and reinduction. A *wrapper verification* system monitors the validity of data returned by the wrapper. If the site changes, the wrapper may extract nothing at all or some data that is not correct. The verification system will detect data inconsistency and notify the operator or automatically launch a wrapper repair process. A *wrapper reinduction* system repairs the extraction rules so that the wrapper works on changed pages.

Figure 1 graphically illustrates the entire life cycle of a wrapper. As shown in the figure, the wrapper induction system takes a set of web pages labeled with examples of the data to be extracted. The output of the wrapper induction system is a wrapper, consisting of a set of extraction rules that describe how to locate the desired information on a Web page. The wrapper verification system uses the functioning wrapper to collect extracted data. It then learns patterns describing the structure of data. These patterns are used to verify that the wrapper is correctly extracting data at a later date. If a change is detected, the system can automatically repair a wrapper by using this structural information to locate examples of data on the new pages and re-running the wrapper induction system with these examples. At the core of these wrapper maintenance applications is a machine learning algorithm that learns structural information about common data fields. In this paper we introduce the algorithm, DataProG, and describe its application to the wrapper maintenance tasks in detail. Though we focus on web applications, the learning technique is not web-specific, and can be used for data validation in general.

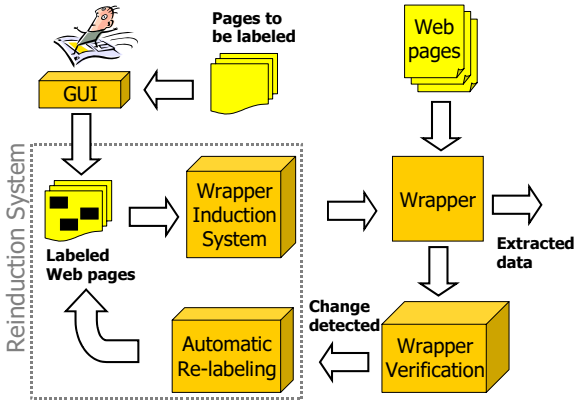


Figure 1: Life cycle of a wrapper

Note that we distinguish two types of extraction rules: landmark-based rules that extract data by exploiting the structure of the Web page, and content-based rules, which we refer to as content patterns or simply patterns, that exploit the structure of the field itself. Our previous work focused on learning landmark rules for information extraction [Muslea *et al.*, 2001]. The current work shows that augmenting these rules with content-based patterns provides a foundation for sophisticated wrapper maintenance applications.

2 Learning Content Patterns

The goal of our research is to extract information from semi-structured information sources. This typically involves identifying small chunks of highly informative data on formatted pages (as opposed to parsing natural language text). Either by convention or design, these *fields* are usually structured: phone numbers, prices, dates, street addresses, names, schedules, *etc.* Several examples of street addresses are given in Fig. 2. Clearly, these strings are not arbitrary, but share some similarities. The objective of our work is to learn the structure of such fields.

4676 Admiralty Way
10924 Pico Boulevard
512 Oak Street
2431 Main Street
5257 Adams Boulevard

Figure 2: Examples of a street address field

2.1 Data Representation

In previous work, researchers described the fields extracted from Web pages by a character-level grammar [Goan *et al.*, 1996] or a collection of global features, such as the number of words and the density of numeric characters [Kushmerick, 1999]. We employ an intermediate word-level representation that balances the descriptive power and specificity of the character-level representation with the compactness and computational efficiency of the global representation. Words, or more

accurately tokens, are strings generated from an alphabet containing different types of characters: alphabetic, numeric, punctuation, *etc.* We use the token's character types to assign it to one or more syntactic categories: alphabetic, numeric, *etc.* These categories form a hierarchy depicted in Fig. 3, where the arrows point from more general to less general categories. A unique specific token type is created for every string that appears in at least k examples, as determined in a preprocessing step. The hierarchical representation allows for multi-level generalization. Thus, the token "Boulevard" belongs to the general token types ALPHANUM (alphanumeric strings), ALPHA (alphabetic strings), UPPER (capitalized words), as well as to the specific type representing the string "Boulevard". This representation is flexible and may be expanded to include domain specific information. For example, the numeric type is divided into categories that include range information about the number — LARGE (larger than 1000), MEDIUM (medium numbers, between 10 and 1000) and SMALL (smaller than 10)— and number of digits: 1−, 2−, and 3−digit. Likewise, we may explicitly include knowledge about the type of information being parsed, *e.g.*, some 5-digit numbers could be represented as ZIPCODE.

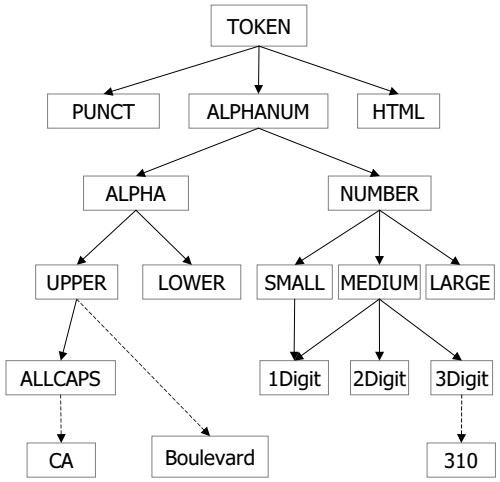


Figure 3: Portion of the token type syntactic hierarchy

We have found that a sequence of specific and general token types is more useful for describing the content of information than the character-level finite state representations used in previous work [Carrasco and Oncina, 1994, Goan *et al.*, 1996]. The character-level description is far too fine grained to compactly describe data and, therefore, leads to poor generality. The coarse-grained token-level representation is more appropriate for most Web data types. In addition, the data representation schemes used in previous work attempt to describe the entire data field, while we use use only the starting and ending sequences, or patterns, of tokens to capture the structure of the data fields. The reason for this is similar to the one above: using the starting and ending patterns allows us to generalize the structural information for many complex fields which have a lot of variability. Such fields, *e.g.*, addresses, usually have some regularity in how they start and end that we can exploit. We call the starting and ending patterns collectively a *data prototype*. As an example, consider a set of street addresses in Fig. 2. All of the examples start with a pattern <NUMBER UPPER> and end with a specific type <Boulevard> or more generally <UPPER>. Note that the pattern language does not allow loops or recursion. We believe that recursive expressions are not useful representations of the types of data we are trying to learn, because they are harder

to learn and lead to over-generalization.

2.2 Learning from Positive Examples

The problem of learning the data prototype from a set of examples that are labeled as belonging (or not) to a class, may be stated in one of two related ways: as a classification or as a conservation task. In the classification task, both positive and the negative instances of the class are used to learn a rule that will correctly classify new examples. Classification algorithms, like FOIL [Quinlan, 1990], use negative examples to guide the specialization of the rule. They construct discriminating descriptions — those that are satisfied by the positive examples and not the negative examples. The conservation task, on the other hand, attempts to find a characteristic description [Dietterich and Michalski, 1981] or conserved patterns [Brazma *et al.*, 1995], in a set of positive examples of a class. Unlike the discriminating description, the characteristic description will often include redundant features. For example, when learning a description of street addresses, with city names serving as negative examples, a classification algorithm will learn that <NUMBER> is a good description, because all the street addresses start with it and none of the city names do. The capitalized word that follows the number in addresses is a redundant feature, because it does not add to the discriminating power of the learned description. However, if an application using this description encounters a zipcode in the future, it will incorrectly classify it as a street address. This problem could have been avoided if <NUMBER UPPER> was learned as a description of street addresses. Therefore, when negative examples are not available to the learning algorithm, the description has to capture all the regularity of data, including the redundant features, in order to correctly identify new instances of the class and differentiate them from other classes. Ideally, the characteristic description learned from positive examples alone is the same as the discriminating description learned by the classification algorithm from positive and negative examples, where negative examples are drawn from infinitely many classes. While most of the widely used machine learning algorithms (*e.g.*, decision trees [Quinlan, 1993], inductive logic programming [Muggleton, 1991]) solve the classification task, there are fewer algorithms that learn characteristic descriptions.

In our applications, an appropriate source of negative examples is problematic; therefore, we chose to frame the learning problem as a conservation task. We introduce an algorithm that learns data prototypes from positive examples of the data field alone. The algorithm finds statistically significant sequences of tokens. A sequence of token types is significant if it occurs more frequently than would be expected if the tokens were generated randomly and independently of one another. In other words, each such sequence constitutes a pattern that describes many of the positive examples of data and is highly unlikely to have been generated by chance.

The algorithm estimates the baseline probability of a token type's occurrence from the proportion of all types in the examples of the data field that are of that type. Suppose we are learning a description of the set of street addresses in Fig. 2, and have already found a significant token sequence — *e.g.*, the pattern consisting of the single token <NUMBER> — and want to determine whether the more specific pattern, <NUMBER UPPER>, is also a significant pattern. Knowing the probability of occurrence of the type UPPER, we can compute how many times UPPER can be expected to follow NUMBER completely by chance. If we observe a considerably greater number of these sequences, we conclude that the longer pattern is also significant.

We use hypothesis testing [Papoulis, 1990] to decide whether a pattern is significant. The null hypothesis is that observed instances of this pattern were generated by chance, via the random, independent generation of the individual token types. Hypothesis testing decides, at a given confidence level, whether the data supports rejecting the null hypothesis. Suppose n identical sequences

have been generated by a random source. The probability that a token type T (whose overall probability of occurrence is p) will be the next type in k of these sequences has a binomial distribution. For a large n, the binomial distribution approaches a normal distribution $P(x, \mu, \sigma)$ with $\mu = np$ and $\sigma^2 = np(1-p)$. The cumulative probability is the probability of observing at least n_1 events:

$$P(k \geq n_1) = \int_{n_1}^{\infty} P(x, \mu, \sigma) dx \tag{1}$$

We use polynomial approximation formulas [Abramowitz and Stegun, 1964] to compute the value of the integral.

The significance level of the test, α, is the probability that the null hypothesis is rejected even though it is true, and it is given by the cumulative probability above. Suppose we set $\alpha = 0.05$. This means that we expect to observe at least n_1 events 5% of the time under the null hypothesis. If the number of observed events is greater, we reject the null hypothesis (at the given significance level), *i.e.*, decide that the observation is significant. Note that the hypothesis we test is derived from observation (data). This constraint reduces the number of degrees of freedom of the test; therefore, we must subtract one from the number of observed events. This also prevents the anomalous case when a single occurrence of a rare event is judged to be significant.

2.3 DataProG Algorithm

We now describe DataProG, the algorithm that finds statistically significant patterns in a set of token sequences. During the preprocessing step the text is tokenized, and the tokens are assigned one or more syntactic types (see Figure 3). The patterns are encoded in a type of prefix tree, where each node corresponds to a token type. DataProG relies on significance judgements to grow the tree and prune the nodes. Every path through the resulting tree starting at the root node corresponds to a significant pattern found by the algorithm. In this section, we focus the discussion on the version of the algorithm that learns starting patterns. The algorithm is easily adapted to learn ending patterns.

We present the pseudocode of the DataProG algorithm in Table 1. DataProG grows the pattern tree incrementally by (1) finding all significant specializations (*i.e.*, longer patterns) of a pattern and (2) pruning the less significant of the generalizations (or specializations) among patterns of the same length. As the last step, DataProG extracts all significant patterns from the pattern tree, including those generalizations (*i.e.*, shorter patterns) found to be significant given the more specific (*i.e.*, longer) patterns.

The tree is empty initially, and children are added to the root node. The children represent all tokens that occur in the first position in the training examples more often than expected by chance. For example, when learning addresses from the examples in Fig. 2, the root will have two child nodes: Alphanum and Number. The tree is extended incrementally at each node Q. A new child is added to Q for every significant specialization of the pattern ending at Q. As explained previously, a child node is judged to be significant with respect to its parent node if the number of occurrences of the pattern ending at the child node is sufficiently large, given the number of occurrences of the pattern ending at the parent node and the baseline probability of the token type used to extend the pattern. To illustrate on our addresses example, suppose we have already found that a pattern <Number Upper> is significant. There are five ways to extend the tree (see Fig. 4) given the data: <Number Upper Alphanum>, <Number Upper Alpha>, <Number Upper Upper>, <Number Upper Street>, <Number Upper Boulevard>, and <Number Upper Way>. All but the last of these patterns are judged to be significant at $\alpha = 0.05$. For example, <Number

DATAPROG MAIN LOOP
Create root node of tree;
For next node Q of tree
    Create children of Q;
    Prune nodes;
Extract patterns from tree;

CREATE CHILDREN OF Q
For each token type T at next position in examples
  Let C = NewNode;
  Let $C.token$ = T;
  Let $C.examples$ = Q.examples that are followed by T;
  Let $C.count = |C.examples|$;
  Let $C.pattern$ = concat($Q.pattern\, T$);
  If Significant($C.count,\ Q.count,\ T.probability$)
      AddChildToTree(C, Q);
  End If
End T loop

PRUNE NODES
For each child C of Q
  For each sibling S of C s.t. $S.pattern \subset C.pattern$
    Let $N = C.count - S.count$
    If Not(Significant($N, Q.count, C.token.probability$))
      Delete C;
      break;
    Else
      Delete S;
    End If
  End S loop
End C loop

EXTRACT PATTERNS FROM TREE
Create empty list;
For every node Q of tree
  For every child C of Q
    Let $N = C.count - \sum_i (S_i.count | S_i \in Children(C))$
    If Significant($N, Q.count, C.token.probability$)
      Add $C.pattern$ to the list;
Return (list of patterns);

Table 1: Pseudocode of the DataProG algorithm

Upper Upper> is significant, because Upper follows the pattern <Number Upper> five out of five times,[1] and the probability of observing at least that many longer sequences purely by chance is 0.0002.[2] Since this probability is less than α, we judge this sequence to be significant.

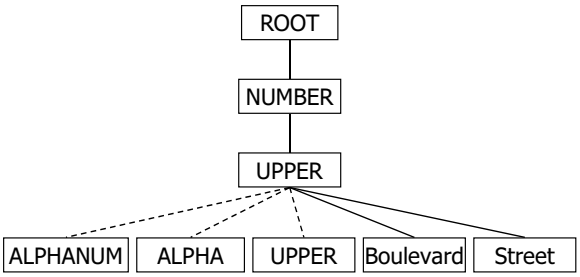


Figure 4: Pattern tree that describes the structure of addresses. Dashed lines link to nodes that are deleted during the pruning step.

The next step is to prune the tree. The algorithm examines each pair of sibling nodes, one of which is more general than the other, and eliminates the less significant of the pair. More precisely, the algorithm iterates through the newly created children of Q, from the most to least general, and for every pair of children C_i and C_j, such that $C_i.pattern \subset C_j.pattern$ (*i.e.*, $C_j.pattern$ is strictly more general than $C_i.pattern$), the algorithm keeps only C_j if it explains significantly more data; otherwise, it keeps only C_i. [3]

Let us illustrate the pruning step with the example pattern tree in Fig. 4. We can eliminate the node AlphaNum, because all the examples that match the pattern <Number Upper Alphanum> also match the pattern <Number Upper Alpha> — thus, Alphanum is not significant given its specialization Alpha. We can eliminate node Alpha for a similar reason. Next, we check whether <Number Upper Upper> is significant given the patterns <Number Upper Boulevard> and <Number Upper Street>. There are 2 instances of the address field that match the pattern <Number Upper Boulevard>, and 2 addresses that match <Number Upper Street>. If <Number Upper Upper> matches significantly more than 4 addresses, it will be retained and the more specific patterns will be pruned from the tree; otherwise, it will be deleted and the more specific ones kept. Because every example is described by at most one pattern of a given length, the pruning step ensures that the size of the tree remains polynomial in the number of tokens, thereby, guaranteeing a reasonable performance of the algorithm.

Once the entire tree has been expanded, the final step is to extract all significant patterns from the tree. Here, the algorithm judges whether the shorter (more general) pattern, *e.g.*, <Number Upper>, is significant given the longer specializations of it, *e.g.*, <Number Upper Boulevard> and <Number Upper Street>. This amounts to testing whether the excess number of examples that

[1]Such small numbers are used for illustrative purposes only — the typical data sets from which the patterns are learned are much larger.

[2]The calculation of this cumulative probability depends on the occurrence probability of Upper. We count the occurrence of each token type independently of the others. In our example, occurrence probability (relative fraction) of type Upper is 0.18.

[3]DataProG is based on an earlier version of the algorithm, DataPro, described in the conference paper [Lerman and Minton, 2000]. Note that in the original version of the algorithm, the specific patterns were always kept, regardless of whether the more general patterns were found to be significant or not. This introduced a strong bias for specific patterns into the results, which led to a high proportion of false positives during the wrapper verification experiments. Eliminating the specificity bias, improved the performance of the algorithm on the verification task.

BUSINESS NAME	ADDRESS
Chado Tea House	8422 West 1st Street
Saladang	363 South Fair Oaks Avenue
Information Sciences Institute	4676 Admiralty Way
Chaya Venice	110 Navy Street
Acorda Therapeutics	330 West 58th Street
Cajun Kitchen	420 South Fairview Avenue
Advanced Medical Billing Services	9478 River Road
Vega 1 Electrical Corporation	1723 East 8th Street
21st Century Foundation	100 East 85th Street
TIS the Season Gift Shop	15 Lincoln Road
Hide Sushi Japanese Restaurant	2040 Sawtelle Boulevard
Afloat Sushi	87 East Colorado Boulevard
Prebica Coffee & Cafe	4325 Glencoe Avenue
L ' Orangerie	903 North La Cienega Boulevard
Emils Hardware	2525 South Robertson Boulevard
Natalee Thai Restaurant	998 South Robertson Boulevard
Casablanca	220 Lincoln Boulevard
Antica Pizzeria	13455 Maxella Avenue
NOBU Photographic Studio	236 West 27th Street
Lotus Eaters	182 5th Avenue
Essex On Coney	1359 Coney Island Avenue
National Restaurant	273 Brighton Beach Avenue
Siam Corner Cafe	10438 National Boulevard
Grand Casino French Bakery	3826 Main Street
Alejo ' s Presto Trattoria	4002 Lincoln Boulevard
Titos Tacos Mexican Restaurant Inc	11222 Washington Place
Killer Shrimp	523 Washington Boulevard
Manhattan Wonton CO	8475 Melrose Place
Starting patterns	
<Alpha Upper>	<Number Upper Upper>
<Alpha Upper Upper Restaurant>	<Number Upper Upper Avenue>
<Alpha '>	<Number Upper Upper Boulevard>

Table 2: Examples of the business name and address fields from the *Bigbook* source, and the patterns learned from them

are explained by the shorter pattern, and not by the longer patterns, is significant. Any pattern that ends at a terminal node of the tree is significant. Note that the set of significant patterns may not cover all the examples in the data set, just a fraction of them that occur more frequently than expected by chance (at some significance level). Tables 2–4 show examples of several data fields from a yellow pages source (*Bigbook*) and a stock quote source (*Yahoo Quote*), as well as the starting patterns learned for each field.

3 Applications of Pattern Learning

As we explained in the introduction, wrapper induction systems use information from the layout of Web pages to create data extraction rules and are therefore vulnerable to changes in the layout, which occur frequently when the site is redesigned. In some cases the wrapper continues to extract, but the data is no longer correct. The output of the wrapper may also change because the format of

CITY	STATE	PHONE
Los Angeles	CA	(323) 655 - 2056
Pasadena	CA	(626) 793 - 8123
Marina Del Rey	CA	(310) 822 - 1511
Venice	CA	(310) 396 - 1179
New York	NY	(212) 376 - 7552
Goleta	CA	(805) 683 - 8864
Marcy	NY	(315) 793 - 1871
Brooklyn	NY	(718) 998 - 2550
New York	NY	(212) 249 - 3612
Buffalo	NY	(716) 839 - 5090
Los Angeles	CA	(310) 477 - 7242
Pasadena	CA	(626) 792 - 9779
Marina Del Rey	CA	(310) 823 - 4446
West Hollywood	CA	(310) 652 - 9770
Los Angeles	CA	(310) 839 - 8571
Los Angeles	CA	(310) 855 - 9380
Venice	CA	(310) 392 - 5751
Marina Del Rey	CA	(310) 577 - 8182
New York	NY	(212) 924 - 7840
New York	NY	(212) 929 - 4800
Brooklyn	NY	(718) 253 - 1002
Brooklyn	NY	(718) 646 - 1225
Los Angeles	CA	(310) 559 - 1357
Culver City	CA	(310) 202 - 6969
Marina Del Rey	CA	(310) 822 - 0095
Culver City	CA	(310) 391 - 5780
Marina Del Rey	CA	(310) 578 - 2293
West Hollywood	CA	(323) 655 - 6030
Starting patterns		
<Upper Upper> <Upper Upper Rey>	<AllCaps>	<(3digit) 3digit - Large>

Table 3: Examples of the city, state and phone number fields from the *Bigbook* source, and the patterns learned from them

PRICE CHANGE	TICKER	VOLUME	PRICE
+ 0 . 51	INTC	17 , 610 , 300	122 3 / 4
+ 1 . 51	IBM	4 , 922 , 400	109 5 / 16
+ 4 . 08	AOL	24 , 257 , 300	63 13 / 16
+ 0 . 83	T	8 , 504 , 000	53 1 / 16
+ 2 . 35	LU	9 , 789 , 300	68
	ATHM	5 , 646 , 400	29 7 / 8
- 10 . 84	COMS	15 , 388 , 200	57 11 / 32
- 1 . 24	CSCO	19 , 135 , 900	134 1 / 2
- 1 . 59	GTE	1 , 414 , 900	65 15 / 16
- 2 . 94	AAPL	2 , 291 , 800	117 3 / 4
+ 1 . 04	MOT	3 , 599 , 600	169 1 / 4
- 0 . 81	HWP	2 , 147 , 700	145 5 / 16
+ 4 . 45	DELL	40 , 292 , 100	57 3 / 16
+ 0 . 16	GM	1 , 398 , 100	77 15 / 16
- 3 . 48	CIEN	4 , 120 , 200	142
+ 0 . 49	EGRP	7 , 007 , 400	25 7 / 8
- 3 . 38	HLIT	543 , 400	128 13 / 16
+ 1 . 15	RIMM	307 , 500	132 1 / 4
	C	6 , 145 , 400	49 15 / 16
- 2 . 86	GPS	1 , 023 , 600	44 5 / 8
- 6 . 46	CFLO	157 , 700	103 1 / 4
- 0 . 82	DCLK	1 , 368 , 100	106
+ 2 . 00	NT	4 , 579 , 900	124 1 / 8
+ 0 . 13	BFRE	149 , 000	46 9 / 16
- 1 . 63	QCOM	7 , 928 , 900	128 1 / 16
Starting patterns			
<Punct 1digit . 2digit>	<AllCaps>	<Number , 3digit , 3digit>	<Medium 1digit / Number> <Medium 15 / 16 >

Table 4: Data examples from the *Yahoo Quote* source, and the patterns learned from them

the source data itself has changed: *e.g.*, when "$" is dropped from the price field ("9.95" instead of "$9.95"), or book availability changes from "Ships immediately" to "In Stock: ships immediately." Because other applications, such as Web agents [Ambite *et al.*, 2002, Chalupsky *et al.*, 2001], rely on data extracted by wrappers, wrapper maintenance is an important research problem. We divide the wrapper maintenance problem into two parts, each described separately in the paper. *Wrapper verification* automatically detects when a wrapper is not extracting data correctly from a Web source, while *wrapper reinduction* automatically fixes broken wrappers. Both applications learn a description of data, of which patterns learned by DataProG are a significant part.

3.1 Wrapper Verification

If the data extracted by the wrapper changes significantly, this is an indication that the Web source may have changed its format. Our wrapper verification system uses examples of data extracted by the wrapper in the past that are known to be correct in order to acquire a description of the data. The learned description contains features of two types: patterns learned by DataProG and global numeric features, such as the density of tokens of a particular type. The application then checks that this description still applies to the new data extracted by the wrapper. Thus, wrapper verification is a specific instance of the data validation task.

The verification algorithm works in the following way. A set of queries is used to retrieve HTML pages from which the wrapper extracts (correct) training examples. The algorithm then computes the values of a vector of features, $\vec{k}$, that describes each field of the training examples. These features include the patterns that describe the common beginnings (or endings) of the field. During the verification phase, the wrapper generates a set of (new) test examples from pages retrieved using the same set of queries, and computes the feature vector $\vec{r}$ associated with each field of the test examples. If the two distributions, $\vec{k}$ and $\vec{r}$ (see Fig. 5), are statistically the same (at some significance level), the wrapper is judged to be extracting correctly; otherwise, it is judged to have failed.

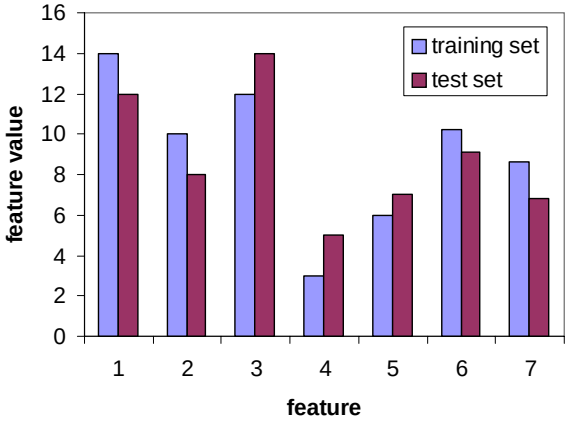


Figure 5: A hypothetical distribution of features over the training and test examples

Each field is described by a vector, whose ith component is the value of the ith feature, such as the number of examples that match pattern j. In addition to patterns, we use the following numeric features to describe the sets of training and test examples: the average number of tuples-per-page, mean number of tokens in the examples, mean token length, and the density of alphabetic,

numeric, HTML-tag and punctuation types. We use goodness of fit method (Papoulis 1990) to decide whether the two distributions are the same. To use the goodness of fit method, we must first compute Pearson's test statistic for the data. The Pearson's test statistic is defined as:

$$q = \sum_{i=1}^{m} \frac{(t_i - e_i)^2}{e_i} \tag{2}$$

where t_i is the observed value of the ith feature in the test data, and e_i is the expected value for that feature, and m is the number of features. For the patterns $e_i = nr_i/N$, where r_i is the number of training examples explained by the ith patter, N is the number of examples in the training set and n is the number of examples in the test set. For numeric features e_i is simply the value of that feature for the training set. The test statistic q has a chi-squared distribution with $m-1$ independent degrees of freedom. If $q < \chi^2(m-1;\alpha)$, we conclude that at significance level α the two distributions are the same; otherwise, we conclude that they are different. Values of χ^2 for different values of α and m can be looked up in a statistics table or calculated using an approximation formula.

In order to use the test statistic reliably, it helps to use as many independent features as possible. In the series of verification experiments reported in [Lerman and Minton, 2000], we used the starting and ending patterns and the average number of tuples-per-page feature when computing the value of q. We found that this method tended to overestimate the test statistic, because the features (starting and ending patterns) were not independent. In the experiments reported in this paper, we use only the starting patterns, but in order to increase the number of features, we added numeric features to the description of data.

3.1.1 Results

We monitored 27 wrappers (representing 23 distinct Web sources) over a period of ten months, from May 1999 to March 2000. The sources are listed in Table 5. For each wrapper, the results of 15–30 queries were stored periodically, every 7–10 days. We used the same query set for each source, except for the *hotel* source, because it accepted dated queries, and we had to change the dates periodically to get valid results. Each set of new results (test examples) was compared with the last correct wrapper output (training examples).

The verification algorithm used DataProG to learn the starting patterns and numeric features for each field of the training examples and made a decision at a high significance level (corresponding to $\alpha = 0.001$) about whether the test set was statistically similar to the training set. If none of the starting patterns matched the test examples or if the data was found to have changed significantly for any data field, we concluded that the wrapper failed to extract correctly from the source; otherwise, if all the data fields returned statistically similar data, we concluded that the wrapper was working correctly.

A manual check of the 438 comparisons revealed 37 wrapper changes attributable to changes in the source layout and data format.[4] The verification algorithm correctly discovered 35 of these changes and made 15 mistakes. Of these mistakes, 13 were *false positives*, which means that the verification program decided that the wrapper failed when in reality it was working correctly. Only two of the errors were the more important *false negatives*, meaning that the algorithm did not

[4]Seven of these were, in fact, internal to the wrapper itself, as when the wrapper was modified to extract "$22.00" instead of "22.00" for the price field. Because these actions were mostly outside of our control, we chose to classify them as wrapper changes.

Source	Type	Data Fields
airport	tuple/list	airport code, name
altavista	list	url, title
Amazon	tuple	book author, title, price, availability, isbn
arrowlist	list	part number, manufacturer, price, status, description, url
Bigbook	tuple	business name, address, city, state, phone
Barnes&Noble	tuple	book author, title, price, availability, isbn
borders	list	book author, title, price, availability
cuisinenet	list	restaurant name, cuisine, address, city, state, phone, link
geocoder	tuple	latitude, longitude, street, city, state
hotel	list	name, price, distance, url
mapquest	tuple	hours, minutes, distance, url
northernlight	list	url, title
parking	list	lotname, dailyrate
Quote	tuple	stock ticker, price, pricechange, volume
Smartpages	tuple	name, address, city, state, phone
showtimes	list	movie, showtimes
theatre	list	theater name, url, address
Washington Post	tuple	taxi price
whitepages	list	business name, address, city, state, phone
yahoo people	list	name, address, city, state, phone
Yahoo Quote	tuple	stock ticker, price, pricechange, volume
yahoo weather	tuple	temperature, forecast
cia factbook	tuple	country area, borders, population, *etc.*

Table 5: List of sources used in the experiments and data fields extracted from them. Source type refers to how much data a source returns in response to a query — a single tuple or a list of tuples. For *airport* source, the type changed from a single tuple to a list over time.

detect a change in the data source. The numbers above result in the following precision, recall and accuracy values:

$$\begin{aligned} P &= \frac{true\ positives}{true\ positives + false\ positives} = 0.73\,, \\ R &= \frac{true\ positives}{true\ positives + false\ negatives} = 0.95\,, \\ A &= \frac{true\ positives + true\ negatives}{positives\ + negatives} = 0.97\,. \end{aligned}$$

These results are an improvement over those reported in [Lerman and Minton, 2000], which produced $P = 0.47, R = 0.95, A = 0.91$. The poor precision value reported in that work was due to 40 false positives obtained on the same data set. We attribute the improvements both to eliminating the specificity bias in the patterns learned by DataProG and to changing the feature set to include only the starting patterns and additional numeric features.[5]

3.1.2 Discussion of Results

Though we have succeeded in significantly reducing the number of false positives, we have not managed to eliminate them altogether. There are a number of reasons for their presence, some of which point to limitations in our approach.

We can split the types of errors into roughly three not entirely independent classes: improper tokenization, incomplete data coverage, and data format changes. The URL field (Table 6) accounted for a significant fraction of the false positives, in large part due to the design of our tokenizer, which splits text strings on punctuation marks. If the URL contains embedded punctuation (as part of the alphanumeric key associated with the user or session id), it will be split into a varying number of tokens, so that it is hard to capture the regularity of the field. The solution is to rewrite the tokenizer to recognize URLs for which well defined specifications exist. We will address this problem in our ongoing work. Our algorithm also failed sometimes (*e.g.*, *arrowlist*, *showtimes*) when it learned very long and specific descriptions. It is worth pointing out, however, that it performed correctly in over two dozen comparisons for these sources. These types of errors are caused by incomplete data coverage: a larger, more varied training data set would produce more general patterns, which would perform better on the verification task. A striking example of the data coverage problem occurred for the stock quotes source: the day the training data was collected, there were many more down movements in the stock price than up, and the opposite was true on the day the test data was collected. As a result, the price change fields for those two days were dissimilar. Finally, because DataProG learns the format of data, false positives will inevitably result from changes in the data format and do not indicate a problem with the algorithm. This is the case for the factbook source, where the units of area changed from "km2" to "sq km".

3.2 Wrapper Reinduction

If the wrapper stops extracting correctly, the next challenge is to rebuild it automatically [Cohen, 1999]. The extraction rules for our wrappers [Muslea *et al.*, 2001], as well as many others (cf. [Kushmerick *et al.*, 1997, Hsu and Dung, 1998]), are generated by a machine learning algorithm, which

[5]Note that this improvement does not result simply from adding numeric features. To check this, we ran the verification experiments on a subset of data (the last 278 comparisons) using only the global numeric features and obtained $P = 0.92$ and $R = 0.55$, whereas using both patterns and numeric features results in values of $P = 0.71$ and $R = 1.00$ for the same data set.

hotel, mapquest (5 cases): URL field contains alphanumeric keys, with embedded punctuation symbols. The tokenizer splits the field into many tokens. The key or its format changes from:

http://...&Stamp=Q4aaiEGSp68*itn/hot%3da11204,itn/agencies/newitn... to
http://...&Stamp=8∼bEgGEQrCo*itn/hot%3da11204,itn/agencies/newitn...
On one occasion, the server name inside the URL changed: from
http://enterprise.mapquest.com/mqmapgend?MQMapGenRequest=... to
http://sitemap.mapquest.com/mqmapgend?MQMapGenRequest=...

showtimes, arrowlist (5 cases): Instance of the showtimes field and part number and description fields (arrowlist) are very long. Many long, overly specific patterns are learned for these fields: *e.g.*,
<(NUMBER : 2DIGIT ALLCAPS) , (SMALL : 2DIGIT) , (SMALL : 2DIGIT) , (4 : 2DIGIT) , 6 : 2DIGIT , 7 : 2DIGIT , 9 : 2DIGIT , 10 : 2DIGIT >

altavista (1 case): Database of the search engine appears to have been updated. A different set of results is returned for each query.

quote (1 case): Data changed — there were many more positive than negative price movements in the test examples

factbook (1 case): Data format changed:
from <NUMBER km2 >
to <NUMBER sq km >

Table 6: List of sources of false positive results on the verification task

takes as input several pages from a source and labeled examples of data to extract from each page. It is assumed that the user labeled all examples correctly. If we label at least a few pages for which the wrapper fails by correctly identifying examples of data on them, we can use these examples as input to the induction algorithm, such as STALKER,[6] to generate new extraction rules.[7] Note that we do not need to identify the data on every page — depending on how regular the data layout is, STALKER can learn extraction rules using a small number of correctly labeled pages. Our solution is to bootstrap the wrapper induction process (which learns landmark-based rules) by learning content-based rules. We want to re-learn the landmark-based rules, because for the types of sites we use, these rules tend to be much more accurate and efficient than content-based rules.

We employ a method that takes a set of training examples, extracted from the source when the wrapper was known to be working correctly, and a set of pages from the same source, and uses a mixture of supervised and unsupervised learning techniques to identify examples of the data field on new pages. We assume that the format of data did not change. Patterns learned by DATAPROG play a significant role in the reinduction task. In addition to patterns, other features, such as the length of the training examples and structural information about pages are used. In fact, because page structure is used during a critical step of the algorithm, we discuss our approach to learning it in detail in the next paragraph.

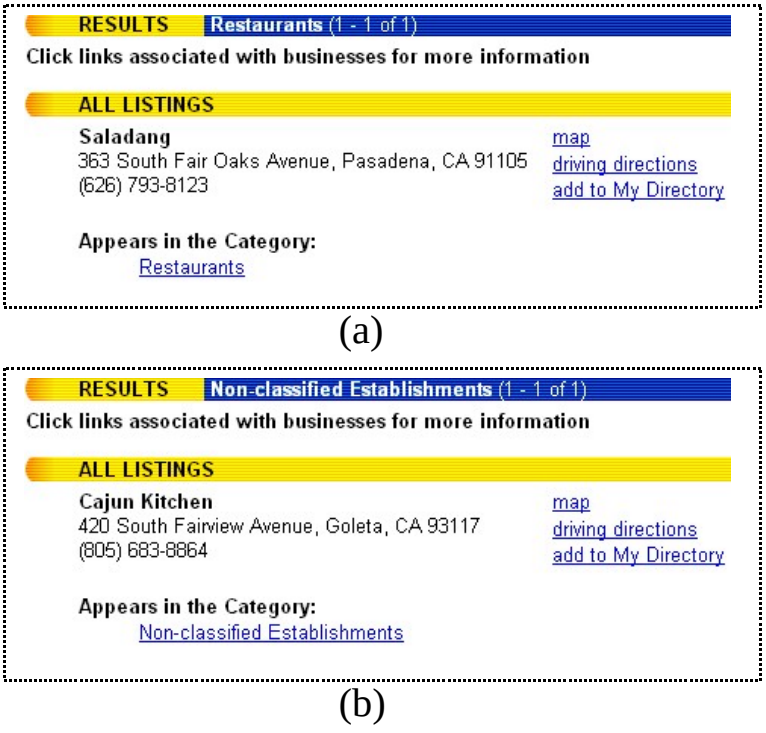


Figure 6: Fragments of two Web pages from the same source displaying restaurant information.

[6]It does not matter, in fact, matter which wrapper induction system is used. We can easily replace STALKER with HLRT [Kushmerick *et al.*, 1997] to generate extraction rules.

[7]In this paper we will only discuss wrapper reinduction for information sources that return a single tuple of results per page, or a detail page. In order to create data extraction rules for sources that return lists of tuples, the STALKER wrapper induction algorithm requires user to specify the first and last elements of the list, as well as at least two consecutive elements. Therefore, we need to be able to identify these data elements with a high degree of certainty.

Page Template Algorithm Many Web sources use templates, or page skeletons, to automatically generate pages and fill them with results of a database query. This is evident in the example in Fig. 6. The template consists of the heading "RESULTS", followed by the number of results that match the query, the phrase "Click links associated with businesses for more information," then the heading "ALL LISTINGS," followed by the anchors "map," "driving directions," "add to My Directory" and the bolded phrase "Appears in the Category." Obviously, data is not part of the template — rather, it appears in the *slots* between template elements.

Given two or more example pages from the same source, we can induce the template used to generate them (Table 7). The template finding algorithm looks for all sequences of tokens — both HTML tags and text — that appear exactly once on each page. The algorithm works in the following way: we pick the smallest page in the set as the template seed. Starting with the first token on this page, we grow a sequence by appending tokens to it, subject to the condition that the sequence appears on every page. If we managed to build a sequence that's at least three tokens long[8], and this sequence appears exactly once on each page, it becomes part of the page template. Templates play an important role in helping identify correct data examples on pages.

```
input:
    P = set of N Web pages
output:
    T = page template
begin
    p = shortest(P)
    T = null
    s = null
    for t = firsttoken(p) to lasttoken(p)
        s' = concat(s, t)
        if ( s' appears on every page in P )
          s = s'
          continue
        else
          n = sum_{page=1}^{N} count(s, page)
          if ( n = N AND length(s) ≥ 3 )
            add-to-template(T, s)
          end if
          s = null
        end if
    end for
end
```

Table 7: Pseudocode of the template finding algorithm

Automatic Labeling Algorithm Figure 7 is a schematic outline of the reinduction algorithm, which consists of automatic data labeling and wrapper induction. Because the latter aspect is described in detail in other work [Muslea *et al.*, 2001], we focus the discussion below on the automatic data labeling algorithm.

[8]The best value for the minimum length for the page template element was determined empirically to be three.

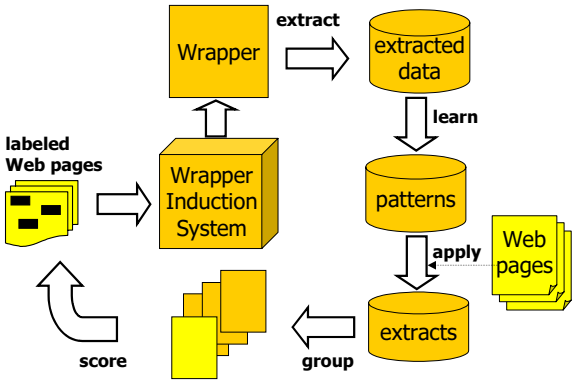


Figure 7: Schematic outline of the reinduction algorithm

First, DataProG learns the starting and ending patterns that describe the set of training examples. These training examples have been collected during wrapper's normal operation, while it was correctly extracting data from the Web source. The patterns are used to identify possible examples of the data field on the new pages. In addition to patterns, we also calculate the mean (and its variance) of the number-of-tokens in the training examples. Each new page is then scanned to identify all text segments that begin with one of the starting patterns and end with one of the ending patterns. Text segments that contain significantly more or fewer tokens than expected based on the old number-of-tokens distribution, are eliminated from the set of candidate extracts. The learned patterns are often too general and will match many, possibly hundreds, text segments on each page. Among these spurious text segments is the correct example of the data field. The rest of the discussion is concerned with identifying the correct examples of data on pages.

We exploit some simple *a priori* assumptions about the structure of Web pages to help us separate interesting extracts from noise. We expect examples of the same data field to appear roughly in the same position and in the same context on each page. For example, Fig. 6 shows fragments of two Web pages from the same source displaying restaurant information. On both pages the relevant information about the restaurant appears after the heading "ALL LISTINGS" and before the phrase "Appears in the Category:". Thus, we expect the same field, *e.g.*, address, to appear in the *same place*, or slot, within the page template. Moreover, the information we are trying to extract will not usually be part of the page template; therefore, candidate extracts that are part of the page template can be eliminated from consideration. Restaurant address always follows restaurant name (in bold) and precedes the city and zip code, *i.e.*, it appears in the *same context* on every page. A given field is either visible to the user on every page, or it is invisible (part of an HTML tag) on every page. In order to use this information to separate extracts, we describe each candidate extract by a feature vector, which includes positional information, defined by the (page template) slot number and context. The context is captured by the adjacent tokens: one token immediately preceding the candidate extract and one token immediately following it. We also use a binary feature which has the value one if the token is visible to the user, and zero if it is part of an HTML tag. Once the candidate extracts have been assigned feature vectors, we split them into groups, so that within each group, the candidate extracts are described by the same feature vector.

The next step is to score groups based on their similarity to the training examples. We expect the highest scoring group to contain correct examples of the data field. One scoring method involves

assigning a rank to the groups based on how many extracts they have in common with the training examples. This technique generally works well, because at least some of the data usually remains the same when the Web page layout changes. Of course, this assumption does not apply to data that changes frequently, such as weather information, flight arrival times, stock quotes, *etc.* However, we have found that even in these sources, there is enough overlap in the data that our approach works. If the scoring algorithm assigns zero to all groups, *i.e.*, there exist no extracts in common with the training examples, a second scoring algorithm is invoked. This scoring method follows the wrapper verification procedure and finds the group that is most similar to the training examples based on the patterns learned from the training examples.

The final step of the wrapper reinduction process is to provide the extracts in the top ranking group to the STALKER wrapper induction algorithm [Muslea *et al.*, 2001] along with the new pages. STALKER learns data extraction rules for the changed pages. Note that examples provided to STALKER are required to be the correct examples of the field. If the set of automatically labeled examples includes false positives, STALKER will not learn correct extraction rules for that field. False negatives are not a problem, however. If the reinduction algorithm could not find the correct example of data on a page, that page is simply not used in the wrapper induction stage.

3.2.1 Results

To evaluate the reinduction algorithm we used only the ten sources (listed in Table 5) that returned a single tuple of results per page, a detail page.[9] The method of data collection was described in Sec. 3.1.1. Over the period between October 1999 and March 2000 there were eight format changes in these sources. Since this set is much too small for evaluation purposes, we created an artificial test set by considering all ten data sets collected for each source during this period. We evaluated the algorithm by using it to extract data from Web pages for which correct output is known. Specifically, we took ten tuples from a set collected on one date, and used this information to extract data from ten pages (randomly chosen) collected at a later date, regardless of whether the source had actually changed or not. We reserved the remaining pages collected at a later date for testing the learned STALKER rules.

The output of the reinduction algorithm is a list of tuples extracted from ten pages, as well as extraction rules generated by STALKER for these pages. Though in most cases we were not able to extract every field on every pages, we can still learn good extraction rules with STALKER as long as few examples of each field are correctly labeled. We evaluated the reinduction algorithm in two stages: first, we checked how many data fields for each source were identified successfully; second, we checked the quality of the learned STALKER rules by using them to extract data from test pages.

Extracting with content-based rules We judged a data field to be successfully extracted if the automatic labeling algorithm was able to identify it correctly on at least two of the ten pages. This is the minimum number of examples STALKER needs to create extraction rules. In practice, such a low success rate only occurred for one field each in two of the sources: *Quote* and *Yahoo Quote*. For all other sources, if a field was successfully extracted, it was correctly identified in at least three, and in most cases almost all, of the pages in the set. A false positive occurred when the reinduction algorithm incorrectly identified some text on a page as a correct example of

[9] We did not use the *geocoder* and *cia factbook* wrappers in the experiments. The *geocoder* wrapper accessed the source through another application; therefore, the pages were not available to us for analysis. The reason for excluding the *factbook* is that it is a plain text source, while our methods apply to Web pages. Note also that in the verification experiments, we had two wrappers for the *mapquest* source, each extracting different data. In the experiments described below, we used the one that contained more data for this time period.

a data field. In many cases, false positives consisted of partial fields, *e.g.*, "Cloudy" rather than "Mostly Cloudy" (*yahoo weather*). A false negative occurred when the algorithm did not identify any examples of a data field. We ran the reinduction experiment attempting to extract the fields listed in Table 8. The second column of the table lists the fractions of data sets for which the field was successfully extracted. We were able to correctly identify fields 277 times across all data sets making 61 mistakes, of which 31 were attributed to false positives and 30 to the false negatives.

There are several reasons the reinduction algorithm failed to operate perfectly. In many cases the reason was the small training set.[10] We can achieve better learning for the yellow-pages-type sources *Bigbook* and *Smartpages* by using more training examples (see Fig. 8). In two cases, the errors were attributable to changes in the format of data, which resulted in the failure of patterns to capture the structure of data correctly: *e.g.*, the *airport* source changed airport names from capitalized words to allcaps, and in the *Quote* source in which the patterns were not able to identify negative price changes because they were learned for a data set in which most of the stocks had a positive price change. For two sources the reinduction algorithm could not distinguish between correct examples of the field and other examples of the same data type: for the *Quote* source, in some cases it extracted opening price or high price for the stock price field, while for the *yahoo weather* source, it extracted high or low temperature, rather than the current temperature. This problem was also evident in the *Smartpages* source, where the city name appeared in several places on the page. In these cases, user intervention or meta-analysis of the fields may be necessary to improve results of data extraction.

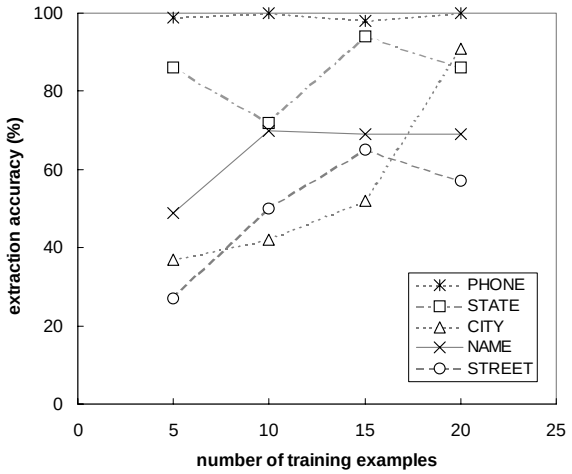


Figure 8: Performance of the reinduction algorithm for the fields in the *Smartpages* source as the size of the training set is increased

Extracting with landmark-based rules The final validation experiment consisted of using the automatically generated wrappers to extract data from test pages. The last three columns in Table 8 list precision, recall and accuracy for extracting data from test pages. The performance is very good for most fields, with the notable exception of the *STATE* field of *Bigbook* source. For that field, the pattern <ALLCAPS> was overly general, and a wrong group received the highest score

[10]Limitations in the data collection procedure prevented us from accumulating large data sets for all sources; therefore, in order to keep the methodology uniform across all sources, we decided to use smaller training sets.

source/field	ex %	p	r
airport code	100	1.0	1.0
airport name	90	1.0	1.0
Amazon author	100	97.3	0.92
Amazon title	70	98.8	0.81
Amazon price	100	1.0	0.99
Amazon ISBN	100	1.0	0.91
Amazon availability	60	1.0	0.86
Barnes&Noble author	100	0.93	0.96
Barnes&Noble title	80	0.96	0.62
Barnes&Noble price	90	1.0	0.68
Barnes&Noble ISBN	100	1.0	0.95
Barnes&Noble availability	90	1.0	0.92
Bigbook name	70	1.0	0.76
Bigbook street	90	1.0	0.87
Bigbook city	70	0.91	0.98
Bigbook state	100	0.04	0.50
Bigbook phone	90	1.0	0.30
mapquest time	100	1.0	0.98
mapquest distance	100	1.0	0.98
Quote pricechange	50	0.38	0.36
Quote ticker	63	0.93	0.87
Quote volume	100	1.0	0.88
Quote shareprice	38	0.46	0.60
Smartpages name	80	1.0	0.82
Smartpages street	80	1.0	0.52
Smartpages city	0	0.68	0.58
Smartpages state	100	1.0	0.70
Smartpages phone	100	0.99	1.0
Yahoo Quote pricechange	100	1.0	0.41
Yahoo Quote ticker	100	1.0	0.98
Yahoo Quote volume	100	1.0	0.99
Yahoo Quote shareprice	80	1.0	0.59
Washington Post price	100	1.0	1.0
Weather temp	40	0.36	0.82
Weather outlook	90	0.83	1.0
average	83	0.90	0.80

Table 8: Reinduction results on ten Web sources. The first column lists the fraction of the fields for each source that were correctly extracted by the pattern-based algorithm. We judged the field to be extracted if the algorithm correctly identified at least two examples of it. The last two columns list precision and recall on the data extraction task using the reinduced wrappers.

source/field	P	R	A	source/field	P	R	A
Amazon author	1.0	1.0	1.0	*Smartpages* name	1.0	0.9	0.9
Amazon title	1.0	0.7	0.7	*Smartpages* street	N/A	0.0	0.0
Amazon price	0.9	0.9	0.9	*Smartpages* city	0.0	0.0	0.0
Amazon ISBN	1.0	0.9	0.9	*Smartpages* state	1.0	0.9	0.9
Amazon availability	1.0	0.9	0.9	*Smartpages* phone	N/A	0.0	0.0
Barnes&Noble author	1.0	0.5	0.5	*Yahoo Quote* pricechange	1.0	0.2	0.2
Barnes&Noble title	1.0	0.8	0.8	*Yahoo Quote* ticker	1.0	0.5	0.5
Barnes&Noble price	1.0	1.0	1.0	*Yahoo Quote* volume	1.0	0.7	0.7
Barnes&Noble ISBN	1.0	1.0	1.0	*Yahoo Quote* shareprice	1.0	0.7	0.7
Barnes&Noble availability	1.0	1.0	1.0				
Quote pricechange	0.0	0.0	0.0	*Quote* volume	1.0	1.0	1.0
Quote ticker	1.0	1.0	1.0	*Quote* shareprice	0.0	N/A	0.0

Table 9: Precision, recall, and accuracy of the learned STALKER rules for the changed sources

during the scoring step of the reinduction algorithm. The average precision and recall values were $P = 0.90$ and $R = 0.80$.

Within the data set we studied, five sources, listed in Table 9, experienced a total of seven changes. In addition to these sources, the *airport* source changed the format of the data it returned, but since it simultaneously changed the presentation of data from a detail page to a list, we could not use this data to learn STALKER rules. Table 9 shows the performance of the automatically reinduced wrappers for the changed sources. For most fields precision P, the more important of the performance measures, is close to its maximum value, indicating that there were few false positives. However, small values of recall indicate that not all examples of these fields were extracted. This result can be traced to a limitation of our approach: if the same field appears in a different context, more than one rule is necessary to extract it from a source. In such cases, we extract only a subset of the examples that share the same context, but ignore the rest of the examples.

As mentioned earlier, we believe we can achieve better performance for the yellow-pages-type sources *Bigbook* and *Smartpages* by using more training examples. Figure 8 shows the effect increasing the size of the training example set on the performance of the automatically generated wrappers for the *Smartpages* source. As the number of training examples goes up, the accuracy of most extracted fields goes up.

Lists We have also applied the reinduction algorithm to extract data from pages containing lists of tuples, and, in many cases, have successfully extracted at least several examples of each field from several pages. However, in order to learn the correct extraction rules for sources returning lists of data, STALKER requires that the first, last and at least two consecutive list elements be correctly specified. The methods presented here cannot guarantee that the required list elements are extracted, unless all the list elements are extracted. We are currently working on new approaches to data extraction from lists [Lerman *et al.*, 2001] that will enable us to use STALKER to learn the correct data extraction rules.

4 Previous Work

There has been a significant amount of research activity in the area of pattern learning. In the section below we discuss two approaches, grammar induction and relational learning, and compare their performance to DATAPROG on tasks in the Web wrapper application domain. In Section 4.2

we review previous work on topics related to wrapper maintenance, and in Section 4.3 we discuss related work in information extraction and wrapper induction.

4.1 Pattern Learning

Grammar induction Several researchers have addressed the problem of learning the structure, or patterns, describing text data. In particular, grammar induction algorithms have been used in the past to learn the common structure of a set of strings. Carrasco and Oncina proposed ALERGIA [Carrasco and Oncina, 1994], a stochastic grammar induction algorithm that learns a regular language from positive examples of the language. ALERGIA starts with a finite state automaton (FSA) that is initialized to be a prefix tree that represents all the strings of the language. ALERGIA uses a state-merging approach [Angluin, 1982, Stolcke and Omohundro, 1994] in which the FSA is generalized by merging pairs of statistically similar (at some significance level) subtrees. Similarity is based purely on the relative frequencies of substrings encoded in the subtrees. The end result is a minimum FSA that is consistent with the grammar.

Goan et al. [Goan *et al.*, 1996] found that when applied to data domains commonly found on the Web, such as addresses, phone numbers, *etc.*, ALERGIA tended to merge too many states, resulting in an over-general grammar. They proposed modifications to ALERGIA, resulting in algorithm WIL, aimed at reducing the number of faulty merges. The modifications were motivated by the observation that each symbol in a string belong to one of the following syntactic categories: NUMBER, LOWER, UPPER and DELIM. When viewed on the syntactic level, data strings contain additional structural information that can be effectively exploited to reduce the number of faulty merges. WIL merges two subtrees if they are similar (in the ALERGIA sense) and also if, at every level, they contain nodes that are of the same syntactic type. WIL also adds a wildcard generalization step in which the transitions corresponding to symbols of the same category that are approximately evenly distributed over the range of that syntactic type (*e.g.*, 0–9 for numerals) are replaced with a single transition corresponding to the type (*e.g.*, NUMBER). Goan *et al.* demonstrated that the grammars learned by WIL were more effective in recognizing new strings in several relevant Web domains.

We compared the performance of WIL to DataProG on the wrapper verification task. We used WIL to learn the grammar on the token level using data examples extracted by the wrappers, not on the character level as was done by Goan *et al.*Another difference from Goan *et al.* was that, whereas they needed on the order of 100 strings to arrive at a high accuracy rate, we have on the order of 20–30 examples to work with. Note that we can no longer apply the wildcard generalization step to the FSA because we would need many more examples to decide whether the token is approximately evenly distributed over that syntactic type. Instead, we compare DataProG against two versions of WIL: one without wildcard generalization (WIL1), and one in which every token in the initial FSA is replaced by its syntactic type (WIL2). In addition to the syntactic types used by Goan *et al.*, we also had to introduce another type ALNUM to be consistent with the patterns learned by DataProG. Neither version of WIL allows for multi-level generalization.

The algorithms were tested on data extracted by wrappers from 26 Web sources on ten different occasions over a period of several months (see Sec. 3.1). Results of 20–30 queries were stored every time. For each wrapper, one data set was used as the training examples, and the data set extracted on the very next date was used as test examples. We used WIL1 and WIL2 to learn the grammar of each field of the training examples and then used the grammar to recognize the test examples. If the grammar recognized more than 80% of the test examples of a data field, we concluded that it recognized the entire data field; otherwise, we concluded that the grammar did not recognize

the field, possibly because the data itself has changed. This is the same procedure we used in the wrapper verification experiments, and it is described in greater detail in Section 3.1.1. Over the period of time covered by the data, there were 21 occasions on which a Web site changed, thereby causing the data extracted by the wrapper to change as well. The precision and recall values for WIL1 (grammar induction on specific tokens) were $P = 0.20$, and $R = 0.81$; for WIL2 (grammar induction on wildcards representing tokens' syntactic categories) the values were $P = 0.55$ and $R = 0.76$. WIL1 learned an overly specific grammar, which resulted in a high rate of false positives on the verification task, while WIL2 learned an overly general grammar, resulting in slightly more false negatives. The recall and precision value of DataProG for the same data were $P = 0.73$ and $R = 1.0$.

Recently Thollard *et al.* [Thollard *et al.*, 2000] introduced MDI, an extension to ALERGIA. MDI has been shown to generate better grammars in at least one domain by reducing the number of faulty merges between states . MDI replaces ALERGIA's state merging criterion with a more global measure that attempts to minimize the Kullback-Leibler divergence between the learned automaton and the training sample while at the same time keeping the size of the automaton as small as possible. It is not clear whether MDI (or a combination of MDI/WIL) will lead to better grammars for common Web data types. We suspect not, because regular grammars capture just a few of the multitude of data types found on the Web. For example, business names, such as restaurant names shown in Table 2 may not have a well defined structure, yet many of them start with two capitalized words and end with the word "Restaurant" — which constitute patterns learned by DataProG.

Relational learning As a sequence of n tokens, a pattern can also be viewed as a non-recursive n-ary predicate. Therefore, we can use a relation-learning algorithm like FOIL [Quinlan, 1990] to learn them. Given a set of positive and negative examples of a class, FOIL learns first order predicate logic clauses defining the class. Specifically, it finds a discriminating description that covers many positive and none of the negative examples.

We used Foil.6 with the no-negative-literals option to learn patterns describing several different data fields. In all cases the closed world assumption was used to construct negative examples from the known objects: thus, for the *Bigbook* source, names and addresses were the negative examples for the phone number class. We used the following encoding to translate the training examples to allow foil.6 to learn logical relations. For each data field, FOIL learned clauses of the form

$$data\ field(A) := P(A)\ followed\_by(A, B)\ P(B)\,, \tag{3}$$

as a definition of the field, where A and B are tokens, and the terms on the right hand side are predicates. The predicate $followed\_by(A, B)$ expresses the sequential relation between the tokens. The predicate $P(A)$ allows us to specify the token A as a specific token (*e.g.*, $John(A)$) or a general type (*e.g.*, Upper(A), Alpha(A)), thus, allowing FOIL the same multi-level generalization capability as DataProG.

We ran Foil.6 on the examples associated with the *Bigbook* (see Tables 2–3). The relational definitions learned by Foil.6 from these examples are shown in Table 10.

In many cases, there were similarities between the definitions learned by FOIL and the patterns learned by DataProG, though clauses learned by FOIL tended to be overly general. Another problem was when given examples of a class with little structure, such as names and book titles, FOIL tended to create clauses that covered single examples, or it failed to find any clauses. In general, the description learned by FOIL depended critically on what we supplied as negative examples of that field. For example, if we were trying to learn a definition for book titles in the

```
*** Warning:  the following definition does not cover 23 tuples in the relation
    NAME(A) := AllCaps(A), followed_by(A,B)
    NAME(A) := Upper(A), followed_by(A,B), Number(B)
    NAME(A) := followed_by(A,B), Venice(B)

    STREET(A) := Large(A), followed_by(A,B)
    STREET(A) := Medium(A), followed_by(A,B), AlphaNum(B)

** Warning:  the following definition does not cover 9 tuples in the relation
    CITY(A) := Los(A)
    CITY(A) := Marina(A)
    CITY(A) := New(A)
    CITY(A) := Brooklyn(A)
    CITY(A) := West(A), followed_by(A,B), Alpha(B)

    STATE(A) := CA(A)
    STATE(A) := NY(A)

    PHONE(A) := ((A)
```

Table 10: Definitions learned by foil.6 for the *Bigbook* source

presence of prices, FOIL would learn that something that starts with a capitalized word is a title. If author names were supplied as negative examples as well, the learned definition would have been different. Therefore, using FOIL in situations where the complete set of negative examples is not known or available, is problematic.

4.2 Wrapper Maintenance

Kushmerick [Kushmerick, 1999] addressed the problem of wrapper verification by proposing an algorithm Rapture to verify that a wrapper correctly extracts data from a Web page. In that work, each data field was described by a collection of global features, such as word count, average word length, and density of types, *i.e.*, proportion of characters in the training examples that are of an HTML, alphabetic, or numeric type. Rapture calculated the mean and variance of each feature's distribution over the training examples. Given a set of queries for which the wrapper output is known, Rapture generates a new result for each query and calculates the probability of generating the observed value for every feature. Individual feature probabilities are then combined to produce an overall probability that the wrapper has extracted data correctly. If this probability exceeds a certain threshold, Rapture decides that the wrapper is correct; otherwise, that it has failed. Kushmerick found that the HTML density alone can correctly identify almost all of the changes in the sources he monitored. In fact, adding other features in the probability calculation significantly reduced algorithm's performance. We compared Rapture's performance on the verification task to our approach, and found that Rapture missed 17 wrapper changes (false negatives) if it relied

solely on the HTML density feature. [11]

There has been relatively little prior work on the wrapper reinduction problem. Cohen [Cohen, 1999] adapted WHIRL, a "soft" logic that incorporates a notion of statistical text similarity, to recognize page structure of a narrow class of pages: those containing simple lists and simple hotlists (defined as anchor-URL pairs). Previously extracted data, combined with page structure recognition heuristics, was used to reconstruct the wrapper once the page structure changed. Cohen conducted wrapper maintenance experiments using original data and corrupted data as examples for WHIRL. However, his procedure for corrupting data was not realistic or representative of how data on the Web changes. Although we cannot at present guarantee good performance of our algorithm on the wrapper reinduction for sources containing lists, we handle the realistic data changes in Web sources returning detail pages.

4.3 Information Extraction

Our system, as used in the reinduction task, is related in spirit to the many information extraction (IE) systems developed both by our group and others in that it uses a learned representation of data to extract information from specific texts. Like wrapper induction systems (see [Muslea *et al.*, 2001, Kushmerick *et al.*, 1997, Freitag and Kushmerick, 2000]), it is domain independent and works best with semi-structured data, *e.g.*, Web pages. It does not handle free text as well as other systems, such as AutoSlog [Riloff, 1993] and Whisk [Soderland, 1999]. Unlike wrapper induction, it does not extract data based on the features that appear near it in text, but rather based on the content of data itself. However, unlike Whisk, which also learns content rules, our reinduction system represents each field independently of the other fields, which can be an advantage, for instance, when a web source changes the order in which data fields appear. Another difference is that our system is designed to run automatically, without requiring any user interaction to label informative examples. In the main part because it is purely automatic, the reinduction system fails to achieve the accuracy of other IE systems which rely on labeled examples to train the system; however, we do not see it as a major limitation, since it was designed to *complement* existing extraction tools, rather than supersede them. In other words, we consider the reinduction task to be successful if it can accurately extract a sufficient number of examples to use in a wrapper induction system. The system can then use the resulting wrapper to accurately extract the rest of the data from the source.

There are many similarities between our approach and that used by the RoadRunner system, developed concurrently with our system and reported recently in [Crescenzi *et al.*, 2001b, Crescenzi *et al.*, 2001a]. The goal of that system is to automatically extract data from Web sources by exploiting similarities in page structure across multiple pages. RoadRunner works by inducing the grammar of Web pages by comparing several pages containing long lists of data. The grammar is expressed at the HTML tag level, so it is similar to the extraction rules generated by Stalker. The RoadRunner system has been shown to successfully extract data from several Web sites. The two significant differences between that work and ours are (i) they do not have a way of detecting changes to know when the wrapper has to be rebuild and (ii) our reinduction algorithm works on detail pages only, while RoadRunner works only on lists. We believe that our data-centric

[11] Although we use a different statistical test and cannot compare the performance of our algorithm to Rapture directly, we doubt that it would outperform our algorithm on our data set if it used all global numeric features. We ran the verification experiment on a subset of the verification data (278 comparisons) using all numeric features, which resulted in $P = 0.92$ and $R = 0.55$. Using the approach described in this paper, on the other hand, lead to $P = 0.71$ and $R = 1.00$ for the same data set.

approach is more flexible and will allow us to extract data from more diverse information sources than the RoadRunner approach that only looks at page structure.

5 Conclusion

In this paper we have described the DataProG algorithm, which learns structural information about a data field from a set of examples of the field. We use these patterns in two Web wrapper maintenance applications: (i) **verification** — detecting when a wrapper stops extracting data correctly from a Web source, and (ii) **reinduction** — identifying new examples of the data field in order to rebuild the wrapper if it stops working. The verification algorithm performed with an accuracy of 97%, much better than results reported in our earlier work [Lerman and Minton, 2000]. In the reinduction task, the patterns were used to identify a large number of data fields on Web pages, which were in turn used to automatically learn Stalker rules for these Web sources. The new extraction rules were validated by using them to successfully extract data from sets of test pages.

There remains work to be done on wrapper maintenance. Our current algorithms are not sufficient to automatically create Stalker rules for sources that return lists of tuples. However, preliminary results indicate [Lerman *et al.*, 2001] that it is feasible to combine information about the structure of data with *a priori* expectations about the structure of Web pages containing lists to automatically extract data from lists and assign it to rows and columns. This is a first step towards automatic Web wrapper generation. Another exciting direction for future work is using the DataProG algorithm to automatically create wrappers for new sources in some domain given there are existing wrappers for other sources in the same domain. For example, we can learn the author, title and price fields for the *AmazonBooks* source, and use them to extract the same fields on the *Barnes&NobleBooks* source. Preliminary results show that this is indeed feasible. Automatic wrapper generation is an important cornerstone of information-based applications, including Web agents.

6 Acknowledgements

We would like to thank Priyanka Pushkarna for carrying out the wrapper verification experiments.

The research reported here was supported in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory under contract/agreement numbers F30602-01-C-0197, F30602-00-1-0504, F30602-98-2-0109, in part by the Air Force Office of Scientific Research under grant number F49620-01-1-0053, in part by the Integrated Media Systems Center, a National Science Foundation (NSF) Engineering Research Center, cooperative agreement number EEC-9529152 and in part by the NSF under award number DMI-0090978. The U.S. Government is authorized to reproduce and distribute reports for Governmental purposes notwithstanding any copy right annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

References

[Abramowitz and Stegun, 1964] M. Abramowitz and I. A. Stegun. *Handbook of mathematical functions with formulas, graphs and mathematical tables*. Applied Math. Series 55. National Bureau

of Standards, Washington, D.C., 1964.

[Ambite *et al.*, 2002] Jose Luis Ambite, Greg Barish, Craig A. Knoblock, Maria Muslea, Jean Oh, and Steven Minton. Getting from here to there: Interactive planning and agent execution for optimizing travel. In *The Fourteenth Innovative Applications of Artificial Intelligence Conference (IAAI-2002), Edmonton, Alberta, Canada*, pages 862–869, San Mateo, July 2002. Morgan Kaufmann.

[Angluin, 1982] Dana Angluin. Inference of reversible languages. *Journal of the ACM*, 29(3):741–765, 1982.

[Brazma *et al.*, 1995] Alvis Brazma, Inge Jonassen, Ingvar Eidhammer, and David Gilbert. Approaches to the automatic discovery of patterns in biosequences. Technical report, Department of Informatics, University of Bergen, 1995.

[Carrasco and Oncina, 1994] Rafael C. Carrasco and Jose Oncina. Learning stochastic regular grammars by means of a state merging method. In Rafael C. Carrasco and Jose Oncina, editors, *Proceedings of the Second International Colloquium on Grammatical Inference and Applications (ICGI94)*, volume 862 of *Lecture Notes on Artificial Intelligence*, pages 139–152. Springer Verlag, Berlin, September 1994.

[Chalupsky *et al.*, 2001] Hans Chalupsky, Yolanda Gil, Craig A. Knoblock, Kristina Lerman, Jean Oh, David V. Pynadath, Thomas A. Russ, and Milind Tambe. Electric elves: Applying agent technology to support human organizations. In *Proceedings of the Thirteenth Annual Conference on Innovative Applications of Artificial Intelligence (IAAI-2001), Seattle, WA*, 2001.

[Cohen, 1999] William W. Cohen. Recognizing structure in web pages using similarity queries. In *Proc. of the 16th National Conference on Artificial Intelligence (AAAI-1999)*, pages 59–66, 1999.

[Crescenzi *et al.*, 2001a] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. Automatic web information extraction in the ROADRUNNER system. In *Proceedings of the International Workshop on Data Semantics in Web Information Systems (DASWIS-2001)*, 2001.

[Crescenzi *et al.*, 2001b] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. ROADRUNNER: Towards automatic data extraction from large web sites. In *Proceedings of the 27th Conference on Very Large Databases (VLDB)*, Rome, Italy, 2001.

[Dietterich and Michalski, 1981] Thomas Dietterich and Ryszard Michalski. Inductive learning of structural descriptions. *Artificial Intelligence*, 16:257–294, 1981.

[Doorenbos *et al.*, 1997] Robert B. Doorenbos, Oren Etzioni, and Daniel S. Weld. A scalable comparison-shopping agent for the world-wide web. In *Proceeding of the First International Confence on Autonomous Agents, Marina del Rey, CA*, 1997.

[Freitag and Kushmerick, 2000] Dayne Freitag and Nicholas Kushmerick. Boosted wrapper induction. In *Proceedings of the 7th Conference on Artificial Intelligence (AAAI-2000)*, pages 577–583. AAAI Press, Menlo Park, CA, July 2000.

[Goan *et al.*, 1996] Terrance Goan, Nels Benson, and Oren Etzioni. A grammar inference algorithm for the world wide web. In *Proceedings of AAAI Spring Symposium on Machine Learning in Information Access, Stanford University, CA*, 1996.

[Hsu and Dung, 1998] Chun-Nan Hsu and Ming-Tzung Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Journal of Information Systems*, 23:521–538, 1998.

[Knoblock *et al.*, 2001a] Craig A. Knoblock, Kristina Lerman, Steven Minton, and Ion Muslea. Accurately and reliably extracting data from the web: A machine learning approach. *Data Engineering Bulletin*, 2001.

[Knoblock *et al.*, 2001b] Craig A. Knoblock, Steve Minton, Jose Luis Ambite, Maria Muslea, Jean Oh, and Martin Frank. Mixed-initiative, multi-source information assistants. In *The Tenth International World Wide Web Conference (WWW10), Hong Kong*, 2001.

[Kushmerick *et al.*, 1997] Nicholas Kushmerick, Dan S. Weld, and Robert B. Doorenbos. Wrapper induction for information extraction. In *Proceedings of the Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 729–737, 1997.

[Kushmerick, 1999] Nicholas Kushmerick. Regression testing for wrapper maintenance. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-1999)*, 1999.

[Lerman and Minton, 2000] Kristina Lerman and Steven Minton. Learning the common structure of data. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-2000)*. AAAI Press, Menlo Park, July 26–30 2000.

[Lerman *et al.*, 2001] Kristina Lerman, Craig A. Knoblock, and Steven Minton. Automatic data extraction from lists and tables in web sources. In *Proceedings of the workshop on Advances in Text Extraction and Mining (IJCAI-2001)*. AAAI Press, Menlo Park, 2001.

[Muggleton, 1991] Stephen Muggleton. Inductive logic programming. *New Generation Computing*, 8:295–318, 1991.

[Muslea *et al.*, 1998] Ion Muslea, Steven Minton, and Craig Knoblock. Wrapper induction for semistructured web-based information sources. In *Proceedings of the Conference on Automated Learning and Discovery (CONALD).*, 1998.

[Muslea *et al.*, 2001] Ion Muslea, Steven Minton, and Craig A. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Autonomous Agents and Multi-Agent Systems*, 4:93–114, 2001.

[Papoulis, 1990] Athanasios Papoulis. *Probability and Statistics*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[Quinlan, 1990] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

[Quinlan, 1993] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[Riloff, 1993] Ellen Riloff. Automatically constructing a dictionary for information extraction tasks. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 811–816, Menlo Park, CA, USA, July 1993. AAAI Press.

[Soderland, 1999] Stephen Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1-3):233–272, 1999.

[Stolcke and Omohundro, 1994] Andreas Stolcke and Stephen Omohundro. Inference of finite-state probabilistic grammars. In *Proceedings of the 2nd Int. Colloquium on Grammar Induction, (ICGI-94)*, pages 106–118, 1994.

[Thollard *et al.*, 2000] Franck Thollard, Pierre Dupont, and Colin de la Higuera. Probabilistic DFA inference using Kullback-Leibler divergence and minimality. In *Proceedings of the 17th International Conf. on Machine Learning*, pages 975–982. Morgan Kaufmann, San Francisco, CA, 2000.

Appendix F:

Automatic Data Extraction from Lists and Tables in Web Sources *†

Kristina Lerman[1], Craig Knoblock[1,2] and Steven Minton[2]
1. Information Sciences Institute
Univ. of Southern California
Marina del Rey, CA 90292-6695
2. Fetch Technologies
{lerman,knoblock,minton}@isi.edu

Abstract

We describe a technique for extracting data from lists and tables and grouping it by rows and columns. This is done completely automatically, using only some very general assumptions about the structure of the list. We have developed a suite of unsupervised learning algorithms that induce the structure of lists by exploiting the regularities both in the format of the pages and the data contained in them. Among the tools used are AutoClass for automatic classification of data and grammar induction of regular languages. The approach was tested on 14 Web sources providing diverse data types, and we found that for 10 of these sources we were able to correctly find lists and partition the data into columns and rows.

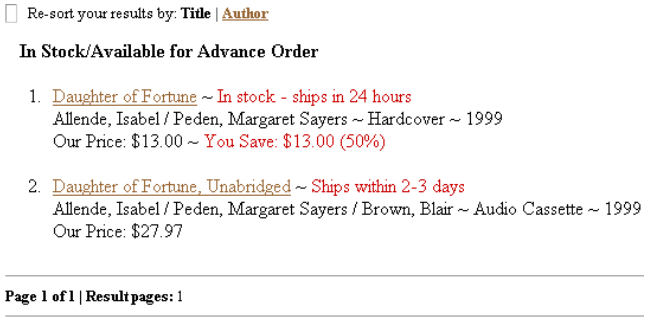
1 Introduction

There is a tremendous amount of information available online, but much of this information is formatted to be easily read by human users, not computer applications. Modern markup languages, like XML, have been advanced to simplify the exchange of information between applications, including software agents; however, XML is not yet in widespread use, it will not help with the legacy data sources not converted to the new standard, and even in the best case it will only address the problem within application domains where all interested parties can agree on the semantic schemas. Until XML becomes ubiquitous, most users will rely on the existing data extraction technologies, the most popular of which are Web wrappers.

A wrapper is a piece of software that turns a Web source into a source that can be queried as if it were a database. The types of sources that this applies to are what are

*Appeared in the Proceedings of the Automatic Text Extraction and Mining Workshop (ATEM-01), IJCAI-01, Seattle, WA, August 2001

called semistructured sources. The pages that come from such sources have no explicit structure or schema, but have an implicit underlying structure. Even text sources such as email messages have some structure in the heading that can be exploited to extract the date, sender, addressee, title, and body of the messages. Other sources, such as online catalogs, have a very regular structure that can be exploited to extract the data. Extraction rules, which the wrapper uses to identify the beginning and end of the data field to be extracted, form an important part of the wrapper. Quick and efficient generation of extraction rules, so-called wrapper induction, has been an active area of research in recent years . The most advanced of such wrapper induction systems use machine learning techniques to learn extraction rules by example. Using a graphical user interface a user marks up data to be extracted on several pages from an online source, and the system generates a set of extraction rules that accurately extract the required information. The wrapper induction system developed at USC is able to efficiently generate extraction rules from a small number of examples. Moreover, it can extract data from pages that contain lists, nested structures, and other complicated formatting layouts.


Re-sort your results by: **Title** | **Author**

In Stock/Available for Advance Order

1. Daughter of Fortune ~ In stock - ships in 24 hours
   Allende, Isabel / Peden, Margaret Sayers ~ Hardcover ~ 1999
   Our Price: $13.00 ~ You Save: $13.00 (50%)

2. Daughter of Fortune, Unabridged ~ Ships within 2-3 days
   Allende, Isabel / Peden, Margaret Sayers / Brown, Blair ~ Audio Cassette ~ 1999
   Our Price: $27.97

Page 1 of 1 | Result pages: 1

Figure 1: A fragment of a Web page containing a list. The page was downloaded from the Borders Books web site.

Wrapper induction for sources containing lists and tables presents a special challenge from the user interface point of view. Consider a Web source, e.g., Borders Books (Figure 1), which provides information about books for sale. The page in the example contains two listings, each consisting of the title, availability, author, format, year of publication, and price of the book. The first listing also contains information about the discount. In order to learn accurate extraction rules for data from this source, the wrapper induction system requires that the user label first, last and at least two consecutive listings. Since this has to be done for several pages, the labeling task quickly becomes tedious and time intensive. However, we have certain expectations about the structure of lists and tables that we can exploit for automatic extraction of data from such sources. For example, each column (e.g., author, availability) of a table usually contains the same type of data, and each row corresponds to a tuple — e.g., (title, availability, author, format, year, price, savings) tuple describing the book — that is repeated in different rows. Moreover, each list from the Borders book site starts after the heading "In Stock/Available for Advance Order" and ends before the string "Page 1 of x—Result pages: x."

In this paper we describe a technique for extracting data from lists and tables and grouping it by rows and columns. This is done completely automatically, using only some very general assumptions about the structure of a list. Lists and tables are alternate ways to present multiple sets of data, and we won't make a distinction between the two. Data on the page is not always laid out in a grid-like pattern; however, it is almost always arranged with a certain amount of regularity we can exploit. For example, in the Borders page containing book listings (see Fig. 1), the title of the book comes first, followed by author, availability, price, followed by the next listing, one listing or tuple per row. Moreover, the tuple elements are arranged in the same order for each listing, so if the author follows the title in one row, it usually follows the title in all other rows. Likewise, we never expect the author in a book listing to appear after the title of the next listing. While these principles may not apply universally, we have found them to be valid for most online sources we studied, including airport listings, online catalogs, hotel directories, *etc.* We tested our approach on 14 Web sources providing diverse types of data and found that for 10 of these sources we were able to correctly find lists and partition the data on the lists into columns and rows.

2 The Approach and Challenges

We have developed a suite of unsupervised learning algorithms that induce the structure of lists by exploiting the regularities both in the format of the pages and the data contained in them. The list below describes the approach at a high level, along with the learning techniques used at each step.

- Extract all data from lists
  - Compute the page template and identify the list on each page
  - Compute a set of features (separators and content) for each data extract
- Identify columns
  Classification of data
- Identify rows
  Grammar-induction on a sequence of class labels

We begin by tokenizing Web pages, that is, splitting the text of the Web pages into individual words or tokens. We analyze pages to find common structure. Many types of Web sources, especially those that return lists, generate the page from a template and fill it with results of a database query. For example, Borders source in the figure above puts book listings after the header "In Stock/Available for Advance Order." By comparing several pages, we are able to deduce the template used to generate them and identify the section of the template that contains the list. Next, we extract all data from the list. If the HTML table has been carefully formatted, this step would amount to extracting all visible text. However, in addition to HTML tags, punctuation characters, such as the tilde in the Borders example, are often used to separate data fields (columns); therefore, we define a column/row separator as a set of sequential HTML tags or any punctuation character excluding the set "(-)'.%" (the choice of the excluded set is discussed in Section 3.1). In the end, the extracted data

are all sequences of tokens that occur between separators. We refer to these sequences of tokens as extracts.

As we mentioned above, we expect all data in the same column to be of the same type, book price for instance, and its content have the same or similar format. In addition to content, layout features, such as separators, may be useful hints for arranging extracts by columns. However, we cannot rely solely on separators — the table may have missing columns, separators before the first row and after the last one may be different from those separating rows within the list, *etc.* Likewise, we cannot rely solely on content — data has a lot of variability, and our representation scheme, like many others, may not be capable of fully capturing distinctions between data. Rather than using each type of evidence separately, we combine them by describing each extract by a set of features that include the separators as well as features that capture the content of data. We use AutoClass to cluster extracts. AutoClass is an unsupervised classification algorithm that finds the optimal number of classes and the best assignment of extracts to classes. In the resulting assignment, each data type ends up in a separate class, or column.

The final step of the analysis is to partition the list into rows. Ideally, it should be easy to identify rows from the class assignment, because a row corresponds to a tuple of data, which is repeated for every row. However, real lists and tables have missing columns, and AutoClass assignment may include errors; therefore, identifying the repeated row pattern is a non-trivial task. We use a grammar induction algorithm for this task. Each list, or rather the sequence of AutoClass-assigned column labels for the extracts in the list, can be thought of as a string generated by a regular language, which we try to learn from the examples of the language. The language captures the repeated structure in the sequences that corresponds to rows. We use this information to partition the list into tuples.

The end result of the application of the suite of algorithms is a complete assignment of data in the list to rows and columns. It is possible to do a meta-analysis of the assignment and fix any errors made along the way, but we have not done so at this point.

3 Algorithms for Automatic Data Extraction

In this section we present details of the algorithms for automatic data extraction. The input is a set of unlabeled Web pages containing lists.

3.1 Finding the page template

During the tokenization step, the text of each Web page is split into individual words, or more accurately tokens, and each token is assigned one or more syntactic types , based on the characters appearing in it. Thus, a token can be an HTML token, an alphanumeric, or a punctuation token. If it's an alphanumeric token, it may also belong to one of two categories: numeric or alphabetic, which is further divided into capitalized or lowercased types, and so on.

Many Web sources use templates to automatically generate pages and fill them with results of a database query. Given two or more example pages from the same source, we can induce the template used to generate them. Our template finding algorithm looks for

all sequences of tokens — both HTML tags and text — that appear exactly once on each page. The algorithm works in the following way: we pick the smallest page in the set as the template seed. Starting with the first token on this page, we grow a sequence by appending tokens to it, subject to the condition that the sequence appears on every page. If we managed to build a sequence that's at least k tokens long,[1] and this sequence appears exactly once on each page, it becomes part of the page template. Figure 2 contains the details of the template finding algorithm.

If any of the lists contains more than two rows, the tags specifying the structure of the list will not be part of the page template, because they will appear more than once on that page. We can use this to our advantage. We look for sections of the page where these sequences of tokens appear more than once, because that's where we expect the list to be. The template finding algorithm has the following pitfall: it can happen that every list starts with exactly the same data; therefore, the beginning of the list will be improperly included in the page template. We can minimize this problem by including diverse pages in the set.

```
input:
        P = set of N Web pages
output:
        T = page template
begin
        p = minimum(P)
        T = null
        s = null
        for t = firsttoken(p) to lasttoken(p)
              s' = concat(s, t)
              if ( s' appears on every page in P )
                s = s'
                continue
              else
                n = sum_{page=1}^{N} count(s, page)
                if ( n = N AND length(s) ≥ 3 )
                  add(s, T)
                end if
                s = null
              end if
        end for
end
```

Figure 2: *Pseudocode of the template finding algorithm*

Once we identify the section of the page that contains the list, we extract all data from it. If the HTML table was carefully formatted, this step would amount to extracting all visible

[1] In our experiments, we have found that $k = 3$ worked best as a minimum length for the page template element.

text. However, in addition to HTML tags, punctuation characters are often used to separate data fields (columns); therefore, we define a column/row separator as a set of sequential HTML tags or any punctuation character excluding the set ".(-)'%". The excluded set was chosen empirically. Sometimes a dash (-) is a good separator, but for many frequently encountered data types, such as phone numbers and zip codes, dash is part of data and not a separator. Likewise, comma (,) is sometimes a separator (*e.g.*, "123 Main St., Pasadena") and sometimes part of data (*e.g.*, in "1,000,000"), though we generally chose to treat it as a separator. In principle, there should be a less *ad hoc* method for choosing separators, which will be the subject of future research. In the end, we extract every sequence of text tokens between separators.

3.2 Identifying columns

We expect all data in the same column to be of the same type, e.g., book prices; therefore, we may be able to identify columns by grouping extracts by similarity of content. In addition to content, layout hints and separators, may be useful evidence for helping arrange extracts by column. However, we cannot rely on either type of evidence by itself. Most methods for representing the content of data, including our own, would be hard pressed to distinguish restaurant names from cities. Likewise, examples of the same data type may contain lots of variability and may be erroneously separated into different columns. While we cannot, for the above reasons, rely on content information when making column assignment, neither can we rely solely on separators. If the table has missing columns, they will affect the separators surrounding visible data. In addition, separators before the first row of the list and after the last row may be different from the ones around the rows within the list. Rather than using either of the two types of evidence alone, we decided to combine them by describing each extract by a set of features that include both the separators and those that capture the content of data.

Each unique separator is assigned an integer. Every extract is described by a set of features, two of which are integers, one for the separator that precedes the extract, and one for the separator that immediately follows it. The content of data is captured by the data prototype, or patterns of specific tokens and syntactic token types that describe the common beginnings and ends of a set of examples of data . For example, a set of street addresses may be well described by the starting pattern "NUMBER CAPS" and the ending pattern "Street", meaning that a significant fraction of addresses start with a number followed by a capitalized word and end in the word "Street." The algorithm (DataPro) that learns the patterns that describe data from positive examples of the field is presented in reference .

Our first approach to computing content features was to use all extracts as examples for DataPro. However, the algorithm tended to overgeneralize by producing patterns that describe more than one column, e.g., addresses and zip codes, which affected the subsequent performance of the classification algorithm. Instead, we adopted a two-step approach, as illustrated in Figure 3. First, we group the extracts by separators, so that the extracts that share at least one separator are in the same cluster. This already does a decent job of separating some of the extracts into columns, though many columns are split among different clusters, and not every extract ends up in a cluster. Extracts within a cluster belong to the

```
input:
        X = set of data extracts from the list
output:
        V = set of vectors describing extracts
begin
        for each x in X
            (V_x)_1 = leftseparator(x)
            (V_x)_2 = rightseparator(x)
        end for
        C = cluster(V)
        P = null
        for each c in C
            addpatterns(patterns(c), P)
        end for

        for each x in X
            for i=firstpattern(P) to lastpattern(P)
                if ( matches(x, patterns(i, P) )
                  (V_x)_{i+2} = 1
                else
                  (V_x)_{i+2} = 0
                end if
            end for
        end for
end
```

Figure 3: *Pseudocode of the algorithm*

same column, and we use the DataPro algorithm to learn the patterns that describe each cluster. Next, we evaluate every extract to see whether it is similar to any cluster. If any of the patterns associated with the nth cluster describe the extract, the value of the nth content feature is one; otherwise, it is zero. Thus, there are as many binary content features as there are clusters.

The two types of evidence are expressed in different units; therefore, standard clustering algorithms that use geometrically based similarity measure (*e.g.*, K-means) would not be appropriate for this purpose. We use AutoClass instead to cluster the extracts. This tool gives us the flexibility to combine different kinds of evidence: class instances may be described by continuous, discrete, and binary values at the same time. AutoClass is a mixture model-based unsupervised classification algorithm that finds both the optimal number of classes and the best (MAP-based) assignment of extracts to classes. It has been used for automatic discovery of classes in diverse data — from DNA to astronomical data sets [**?**]. In the resulting assignment, each data type ends up in a separate class, or column. Because it starts from a random initial assignment which serves as a starting point for the search for both the optimal number of classes and the model that explains the distribution of instance values in each class, AutoClass does not always converge on the same model of data. Thus, it is necessary to run AutoClass several times from different initial random assignments, and choose the outcome that corresponds to the greatest classification likelihood.

3.3 Identifying rows

Finally, we want to find associations between the columns of data by assigning data to tuples. If we anticipate using the Web source in the future, it is more efficient to build a wrapper for it, rather than analyze pages each time information from the source is required. In order to build a wrapper, we need to label the first, last and several consecutive elements of the list. The easiest way to guarantee that the required elements are labeled is to label every element of the list. It is for this reason that we must break the list into rows.

Ideally, it should be easy to identify rows from the column assignment, because each row corresponds to a tuple of data types, and the tuple is expected to be repeated in every row. However, real lists and tables have missing columns, additionally, AutoClass assignment may include errors; therefore, identifying rows is a non-trivial task. We use a grammar induction algorithm to find the repeated cycles of column assignments that correspond to rows. If the extracts are arranged sequentially as they appear in the list, the sequence of their AutoClass-assigned column labels forms a string in a language generated by a regular grammar. Our objective is to learn this grammar from the example strings and to use it to recognize the rows of the list.

Grammar induction, especially the identification of regular languages, has received a great amount of attention in the past three decades . Carrasco and Oncina proposed an algorithm ALERGIA to learn grammars of stochastic regular languages using a state-merging method. The advantages of ALERGIA is that it learns from a set of positive examples of the languages alone; moreover, its performance is polynomial, and indeed has been shown to be linear , in the size of the example set. However, when there are few examples, ALERGIA tends to produce overly complicated grammars, because statistical

significance judgments it uses to learn the grammar are less reliable for small data sets. We adapt a simplified version of ALERGIA to learn the regular grammars associated with lists (Figure 4). Like ALERGIA, we start by constructing a prefix-tree acceptor from the example strings and proceed by merging pairs of equivalent nodes until we arrive at the minimum finite state automaton (FSA) consistent with the language. We examine nodes in the same lexicographic order as ALERGIA; however, unlike ALERGIA, we merge two nodes, i and j, if their incoming arcs, $\delta_{k,i}(a)$ and $\delta_{l,j}(a)$, correspond to the same symbol a and at least one of the outgoing arcs, $\delta_{i,k}(b)$ and $\delta_{j,l}(b)$, from each node correspond to the same symbol. We add an another level of generalization by merging two nodes if they have an incoming arc with the same symbol, and one node is a parent of the other, thereby creating a loop. After a pair of states is merged, we determinize the FSA by making sure all descendents have at most one outgoing transition corresponding to a given symbol. The motivation for the state merging approach is the following: if a column B follows column A and precedes column C in one row, it is likely to follow the same pattern in other rows. Therefore, observing a sequence ABC more than once constitutes evidence that it forms a pattern for a row.

Finally, we extract all cycles from the merged FSA, where each cycle corresponds to a sequence of columns that could constitute a row. This step consists of finding all closed paths through the FSA, *i.e.*, paths that start and end at the same node. Loops, or states that accept a symbol, let's say A, but stay in the same state are represented as A* in the cycle. As in a regular language, this expression means that the symbol A may be repeated any number of times in the language.

To partition the list into rows, we begin at the first symbol of the sequence and find the longest cycle that matches the sequence starting with that symbol. After we find a match, we move to the first unmatched symbol and repeat the procedure for the remaining part of the sequence. Below are the tuples automatically extracted from the list shown in Figure 1 after applying the algorithm to several pages from the Borders book site.

```
(b)Daughter of Fortune
(a)In stock - ships in 24 hours
(c)Allende , Isabel
(c)Peden , Margaret Sayers
(f)Hardcover
(g)1999
(d)Our Price
(e)13 .  00
(h)You Save
(i)13 .  00 ( 50 % )

(b)Daughter of Fortune , Unabridged
(a)In stock - ships in 24 hours
(c)Allende , Isabel
(c)Peden , Margaret Sayers
(c)Brown , Blair
(f)Audio Cassette
(g)1999
```

```
input:
        S: set of example strings
output:
        minimum FSA
begin
        A = prefix tree acceptor from S
        for j = successor(firstnode(A)) to lastnode(A)
            for i = firstnode(A) to j
                if compatible(i, j)
                  merge(A, i, j)
                  determinize(A)
                  exit(i-loop)
                end if
            end for
        end for
        return A
end main


COMPATIBLE(i, j)
input:
        i, j nodes
output:
        boolean
begin compatible
        δ_{k,i} = arc from node k to node i
        δ_{i,m} = arc from node i to node m
        if symbol(δ_{k,i}) = symbol(δ_{l,j}) for some k and l
          if symbol(δ_{i,m}) = symbol(δ_{j,n}) for some m and n
            return true
          end if
          if i is parent of j
            return true
          end if
        end if
end compatible


DETERMINIZE(A)
input:
        FSA A
begin determinize
        for i = firstnode(A) to lastnode(A)
              for j = firstsuccessor(i) to lastsuccessor(i)
                    for ( k = nextsuccessor(j) to lastsuccessor(i)
                          if symbol(δ_{i,j}) = symbol(δ_{i,k})
                            merge(j, k)
                            determinize(A)
                          end if
```

(d)Our Price
(e)27 . 97

4 Results

We have validated our approach by applying it to extract data from 14 Web information sources containing a wide variety of data types. We randomly selected three or four pages from each source, with the only requirement being that the pages contain a list with at least two elements. We applied the extraction algorithm to each set of pages and manually checked whether the data from the list was partitioned correctly into tuples. The table in Fig. 5 summarizes the results. The approach worked for 10 of the 14 sources, though for one source, Yahoo stock quotes, the algorithm made a mistake with two of the 20 tuples. Conceivably, a meta-analysis of the final tuple assignments would be able to catch and correct any errors contained in the preceding steps.

source	pages	extracts	columns	classes	result
airport	4	370	4	5	correct tuples
airport code, location					
Blockbuster	4	663		11	no tuples extracted
movies					
Borders	4	186	9	9	correct tuples
books					
Cuisinenet	3	535	17	15	no tuples extracted
restaurants					
RestaurantRow	4	273	14	14	correct tuples
Yahoo people	3	126	8	8	correct tuples
whitepages					
Yahoo quote	3	259	13	13	18/20 tuples correct
stocks					
Whitepages	3	73	9	5	correct tuples
MapQuest	3	83	5	5	tuples begin in the
driving directions					middle of the rows
hotel	4	163	6	6	correct tuples
CitySearch	4	204	4	6	correct tuples
restaurants					
car rental	4	161	8+	9	correct tuples
boston	4	174	4+	6	correct tuples
restaurants					
Arrow	3	366		10	no tuples extracted
electronic components					

Figure 5: Results of applying the automatic data extraction algorithm to different Web sources.

Our approach failed in four cases for the following reasons. In one case (MapQuest) all three lists began with the same data; therefore, the page template finding algorithm did not locate the correct start of list. In the three other cases, the approach failed because the structure of the list and the resulting FSA for these sources was too complex to extract cycles. Our approach to breaking the list into rows by using grammar induction is clearly not sufficient, and another approach or a modification of the grammar induction algorithm is warranted.

5 Discussion

We have demonstrated that it is possible to accurately extract data from semistructured Web pages containing lists and tables by exploiting the regularities both in the format of the pages and the data contained in them. We have presented a suite of algorithms that extract data from lists and tables and automatically assign it to rows and columns. First, we assign extracts to columns using AutoClass, an unsupervised classification algorithm, then a grammar induction algorithm finds repeated patterns in the column assignments that correspond to rows. The grammar induction may also correct some of the mistakes made by the classification algorithm. It is possible to further analyze the assignment of extracts to tuples to correct mistakes not caught in the preceding steps; however, we have not done this step. We have been able to extract and label data from lists with high accuracy for 10 out of the 14 sources to which we have applied our algorithm. Therefore, we can easily create wrappers for these sources.

One limitation of our approach is that it requires several pages to be analyzed before data can be extracted from a single list. Often, we may have just a single page from a source. Conceivably, there is enough structure in a single list for us to exploit for the purposes of extraction. We are currently considering the extensions of our algorithm that will enable us to extract data from a single list.

Acknowledgements

The research reported here was supported in part by the Rome Laboratory of the Air Force Systems Command and the Defense Advanced Research Projects Agency (DARPA) under contract number F30602-98-2-0109 and by the Air Force Office of Scientific Research under Grant Number F49620-01-1-0053.